



NASA TECHNICAL STANDARD

NASA-STD-4009

**National Aeronautics and Space Administration
Washington, DC 20546-0001**

**Approved: 06-05-2014
Superseding NASA/TM—2010-216809**

**SPACE TELECOMMUNICATIONS RADIO SYSTEM (STRS)
ARCHITECTURE STANDARD**

**MEASUREMENT SYSTEM IDENTIFICATION:
None.**

NASA-STD-4009

DOCUMENT HISTORY LOG

Status	Document Revision	Approval Date	Description
Baseline		06-05-2014	NASA-STD-4009 is based on NASA/TM—2010-216809.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

FOREWORD

This Standard is published by the National Aeronautics and Space Administration (NASA) to provide uniform engineering and technical requirements for processes, procedures, practices, and methods that have been endorsed as standard for NASA programs and projects, including requirements for selection, application, and design criteria of an item.

This Standard is approved for use by NASA Headquarters and NASA Centers, including Component Facilities and Technical and Service Support Centers.

This Standard establishes a description of an architecture standard for NASA space communication radio transceivers. This architecture is a required standard for communication transceiver developments among NASA space missions. Although the architecture was defined to support space-based platforms, the architecture may also be applied to ground station radios.

This Standard strives to provide commonality among NASA radio developments to take full advantage of emerging software-defined radio technologies from mission to mission. This architecture serves as an overall framework for the design, development, operation, and upgrade of these software-based radios.

Requests for information, corrections, or additions to this Standard should be submitted via “Feedback” in the NASA Standards and Technical Assistance Resource Tool at <http://standards.nasa.gov>.

Original Signed By:

Ralph R. Roe, Jr.
NASA Chief Engineer

06-05-2014

Approval Date

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

TABLE OF CONTENTS

<u>SECTION</u>	<u>PAGE</u>
DOCUMENT HISTORY LOG	2
FOREWORD	3
TABLE OF CONTENTS	4
LIST OF FIGURES	7
LIST OF TABLES	8
1. SCOPE	10
1.1 Purpose.....	10
1.2 Executive Summary	11
1.2.1 Key Architecture Requirements.....	11
1.2.2 STRS Overview	12
1.2.3 Roles and Responsibilities	13
1.2.4 Background.....	15
1.3 Applicability	16
1.4 Tailoring.....	16
2. APPLICABLE DOCUMENTS.....	16
2.1 General.....	16
2.2 Government Documents	17
2.3 Non-Government Documents	17
2.4 Order of Precedence.....	17
3. ACRONYMS AND DEFINITIONS.....	17
3.1 Acronyms and Abbreviations	17
3.2 Definitions	20
4. HARDWARE ARCHITECTURE.....	28
4.1 Generalized Hardware Architecture and Specification	29
4.1.1 Components	32
4.1.2 Functions.....	32
4.1.3 Interfaces.....	33
4.1.3.1 External Interfaces	33
4.1.3.2 Networking	34
4.1.3.3 Internal Interfaces	35
4.2 Module Type Specification.....	36
4.2.1 General-Purpose Processing Module.....	36
4.2.1.1 GPM Components.....	36
4.2.1.2 GPM Functions	38
4.2.1.3 GPM Interfaces	38
4.2.1.4 GPM Requirements.....	38
4.2.2 Signal-Processing Module	39
4.2.2.1 SPM Components	40

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

TABLE OF CONTENTS (Continued)

<u>SECTION</u>	<u>PAGE</u>
4.2.2.2 SPM Functions.....	41
4.2.2.3 SPM Interfaces.....	42
4.2.3 Radio Frequency Module.....	43
4.2.3.1 RFM Functions	44
4.2.3.2 RFM Components.....	45
4.2.3.3 RFM Interface.....	45
4.2.3.4 RFM Requirements.....	45
4.2.4 Security Module	45
4.2.5 Networking Module.....	46
4.2.6 Optical Module	46
4.3 Hardware Interface Description.....	46
4.3.1 Control and Data Interface.....	48
4.3.2 DC Power Interface	48
4.3.3 Thermal Interface and Power Consumption	49
5. APPLICATIONS	49
5.1 Application Implementation	49
5.2 Application Selection.....	50
5.3 Navigation Services	50
5.4 Application Repository Submissions.....	51
6. CONFIGURABLE HARDWARE DESIGN ARCHITECTURE.....	52
6.1 Specialized Hardware Interfaces	53
7. SOFTWARE ARCHITECTURE.....	55
7.1 Software Layer Interfaces.....	55
7.2 Infrastructure.....	63
7.3 STRS APIs.....	64
7.3.1 STRS Application-Provided Application Control API.....	64
7.3.2 STRS Infrastructure-Provided Application Control API.....	81
7.3.3 STRS Infrastructure Application Setup API.....	89
7.3.4 STRS Infrastructure Data Sink	94
7.3.5 STRS Infrastructure Data Source	95
7.3.6 STRS Infrastructure Device Control API	96
7.3.7 STRS Infrastructure File Control API	102
7.3.8 STRS Infrastructure Messaging API	107
7.3.9 STRS Infrastructure Time Control API.....	110

TABLE OF CONTENTS (Continued)

<u>SECTION</u>	<u>PAGE</u>
7.3.10	STRS Predefined Data 115
7.3.11	Error Handling 118
7.4	Portable Operating System Interface 118
7.4.1	STRS Application Environment Profile 119
7.5	Network Stack..... 122
7.6	Operating System..... 122
7.7	Hardware Abstraction Layer..... 123
8.	EXTERNAL COMMAND AND TELEMETRY INTERFACES 126
9.	CONFIGURATION FILE (S) 129
9.1	General Configuration File Format Definition and Use 129
9.2	Platform Configuration Files 132
9.3	Application Configuration Files 133

APPENDICES

A	Example Configuration Files 136
A.1	STRS Platform Configuration File Hardware Example 136
A.2	STRS Platform Configuration File Software Example..... 138
A.3	STRS Application Configuration File Example 141
B	POSIX API Profile..... 145
C	Reference Documents 151
D	Acknowledgments 154

LIST OF FIGURES

<u>FIGURE</u>		<u>PAGE</u>
1	Roles and Responsibilities	14
2	Hardware Architecture Diagram Key	30
3	Notional STRS Hardware Architecture Implementation	31
4	GPM Architecture Details	37
5	SPM Architecture Details	40
6	RFM Architecture Details	44
7	Waveform Component Instantiation	50
8	Notional High-Level Software and Configurable Hardware Design Waveform Application Interfaces	54
9	STRS Software Execution Model	57
10	STRS Layered Structure in UML	58
11	STRS Operating Environment	60
12	POSIX-Compliant Versus POSIX-Conformant OS	62
13	STRS Infrastructure	63
14	STRS Application and Device Structure	65
15	STRS Application State Diagram	69
16	Profile Building Blocks	120
17	Command and Telemetry Interfaces	126
18	XML Transformation and Validation	131
19	Configuration File Development Process	132
20	Example of Hardware Portion of STRS Platform Configuration File	136
21	Example of Software Portion of STRS Platform Configuration File	138
22	Example of STRS Waveform Configuration File	142

LIST OF TABLES

<u>TABLE</u>		<u>PAGE</u>
1	STRS Module Interface Characterization.....	47
2	Example—DC Power Interface (Platform Supplied)	49
3	STRS Architecture Subsystem Key	59
4	STRS Software Component Descriptions.....	61
5	APP_Configure()	70
6	APP_GroundTest()	71
7	APP_Initialize()	72
8	APP_Instance()	73
9	APP_Query()	74
10	APP_Read()	75
11	APP_ReleaseObject()	76
12	APP_RunTest()	77
13	APP_Start()	78
14	APP_Stop()	79
15	APP_Write()	80
16	STRS_Configure()	82
17	STRS_GroundTest()	83
18	STRS_Initialize()	84
19	STRS_Query()	85
20	STRS_ReleaseObject()	86
21	STRS_RunTest().....	87
22	STRS_Start().....	88
23	STRS_Stop().....	88
24	STRS_AbortApp()	90
25	STRS_GetErrorQueue().....	90
26	STRS_HandleRequest().....	91
27	STRS_InstantiateApp().....	92
28	STRS_IsOK().....	93
29	STRS_Log()(.....	93
30	STRS_Write()	95
31	STRS_Read()	96
32	STRS_DeviceClose().....	97
33	STRS_DeviceFlush()	97
34	STRS_DeviceLoad().....	98
35	STRS_DeviceOpen()	98
36	STRS_DeviceReset()	99
37	STRS_DeviceStart()	99
38	STRS_DeviceStop().....	100

NASA-STD-4009

LIST OF TABLES (Continued)

<u>TABLE</u>		<u>PAGE</u>
39	STRS_DeviceUnload()	100
40	STRS_SetISR()	101
41	STRS_FileClose()	102
42	STRS_FileGetFreeSpace()	103
43	STRS_FileGetSize()	103
44	STRS_FileGetStreamPointer()	104
45	STRS_FileOpen()	105
46	STRS_FileRemove()	106
47	STRS_FileRename()	106
48	STRS_QueueCreate()	108
49	STRS_QueueDelete()	109
50	STRS_Register()	109
51	STRS_Unregister()	110
52	STRS_GetNanoseconds()	111
53	STRS_GetSeconds()	111
54	STRS_GetTime()	112
55	STRS_GetTimeWrap()	113
56	STRS_SetTime()	113
57	STRS_Synch()	114
58	STRS Predefined Data	115
59	Replacements for Unsafe Functions	122
60	Sample HAL Documentation	125
61	Suggested Services Implemented by the STRS Command and Telemetry Interfaces	128
62	POSIX Subset Profiles PSE51, PSE52, and PSE53	145

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

SPACE TELECOMMUNICATIONS RADIO SYSTEM (STRS) ARCHITECTURE STANDARD

1. SCOPE

This Standard describes the Space Telecommunications Radio System (STRS) architecture for software-defined radios (SDRs), an open architecture for NASA space and ground radios. STRS provides a common, consistent framework to abstract the application software from the radio platform hardware to reduce the cost and risk of using complex reconfigurable and reprogrammable radio systems across NASA missions. It achieves this objective by defining an architecture to enable the reuse of applications (waveforms and services implemented on the SDR) across heterogeneous SDR platforms and reduce dependence on a single vendor. The Standard provides a detailed description and set of requirements to implement the architecture. The Standard focuses on the key architecture components and subsystems by describing their functionality and interfaces for both the hardware and the software, including the applications. The intended audience for this Standard is composed of software, configurable hardware design, and hardware developers who require architecture specification details to develop an SDR platform or application.

A corresponding NASA technical handbook, NASA-HDBK-4009, Space Telecommunications Radio System (STRS) Architecture Standard Rationale, provides the rationale for the decisions made to develop the architecture, provides additional information to clarify the requirements, gives further examples, and answers questions from users.

This Standard is only one of a set of documents to be provided by the mission and used by the STRS platform providers or STRS application developers in the development of an STRS-compliant radio and/or applications. Typical radio acquisition specifications, which include size, weight, power, radiation requirements, connector details, performance and behavior requirements, documentation, and data rights agreements are to accompany this Standard in a radio procurement.

1.1 Purpose

The purpose of this Standard is to establish an open architecture specification for NASA space and ground SDRs. Currently most missions either use hardware radios, which cannot be modified once deployed, or software defined radios with an architecture that requires dependence on the radio provider and significant effort to add new applications. The development of the Standard is part of the larger STRS program currently underway to define NASA's application of software-defined, reconfigurable technology to meet future space communications and navigation system needs. Software-based SDRs enable advanced operations that potentially reduce mission life-cycle costs for space or ground platforms.

SDR technology allows radios to be reconfigured to perform different functions without the necessity of using multiple radios to accomplish each communication function, enabling radio count reduction to reduce mass and power resources.

The STRS project provides the infrastructure and guidance for a repository of applications developed for SDRs using the Standard. Adherence to the Standard for the development of SDR platforms and applications and submittal of the applications to the repository will enable the missions to leverage earlier efforts by reusing various software components compliant with the architecture developed in other NASA programs. This will reduce the cost and risk of deploying SDRs for future NASA missions.

The hardware, configurable hardware design, and software architecture and the supporting documentation defined by the STRS Standard provides the ability to port applications among heterogeneous platforms with minimal effort, reduces the reliance on the initial STRS platform providers, and enables the implementation of the services that are envisioned for NASA radios.

1.2 Executive Summary

1.2.1 Key Architecture Requirements

The key requirements in the development of the STRS architecture are to decrease the development time, cost, and risk of using SDRs while still accommodating advances in technology. The advent of software-based applications allows minimal rework to reuse applications and to adapt to evolving requirements. The architecture does not include mission-specific functional and performance requirements, such as contents or format of the external interfaces to the SDR; waveform-specific requirements such as data rate, coding scheme, and modulation and demodulation techniques; specific hardware; or security, fault tolerance, redundancy, and fault mitigation approaches. Instead the architecture is careful to enable all solutions that the mission might require as they relate to the mission-specific functional and performance specifications.

The requirements for the architecture are derived from the following STRS goals and objectives:

- Usable across most NASA mission types (scalability and flexibility).
- Decrease development time and cost.
- Increase reliability of SDRs.
- Accommodate advances in technology with minimal rework (extensibility).
- Adaptable to evolving requirements (adaptability).
- Leverage existing or developing standards, resources, and experience (state-of-the-art and state-of-practices).
- Maintain vendor independence.
- Enable waveform application portability.

To meet these goals and objectives, the STRS architecture has an open architecture design that accommodates the range of radio form factors that are envisioned by NASA for all mission classes.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

The architecture can also not preclude the implementation of mission-developed services on the SDR such as:

- Multiple waveforms operating simultaneously across any RF band defined in the SDR specification.
- Commanded built-in-test (BIT) and status reporting
- Real-time operational diagnostics
- Automated system recovery and initialization
- Networking and navigation within the SDR
- Secure transmission
- Sharing of processor among on-board elements

1.2.2 STRS Overview

The STRS Standard consists of hardware, configurable hardware design, and software architectures with accompanying description, guidance, and requirements. The hardware architecture is defined in section 4. Section 5 outlines the process and requirements associated with application development. The configurable hardware design architecture is defined in section 6. The software architecture is defined in section 7. An overview of each is provided below.

The terms “software” and “configurable hardware design” are used in this standard to distinguish the architecture items that apply to code (source code, object code, executables, etc.) implemented on a processor; and designs (hardware description language (HDL) source, loadable files, data tables, etc.) implemented in a configurable hardware device such as a field programmable gate array (FPGA). Both items can change the functionality of the radio in-situ using program control. The term “software” is also used in a generic sense in the Standard to discuss all configurable items of the radio, including configurable hardware design. The terminology used is not meant to imply design and implementation process.

The STRS hardware architecture is specified in a modular fashion at a functional level. The hardware architecture standard requires that the hardware provider define the functional breakdown (modules) of the system and publish the functions and interfaces for each module and for the entire radio platform in a hardware interface description (HID) document. Using this information enables NASA and others developing applications or additional modules, or interfacing to the platform, to have the knowledge to integrate and test the hardware interfaces and understand the features and limitations of the platform.

NASA-STD-4009

This Standard encourages the development of applications that are modular, portable, reconfigurable, and reusable. STRS applications use the STRS infrastructure-provided application program interfaces (APIs) and services to load, verify, execute, change parameters, terminate, or unload an application. The STRS applications are submitted to the NASA STRS application repository to allow applications to be reused in the future according to any accepted release agreements. The appropriate application artifacts are submitted to the STRS application repository to provide future missions the information to use the application with limited effort.

The configurable hardware design architecture provides guidance to the development of applications that are partially or fully implemented in a hardware device, such as an FPGA. Early consideration to enable reuse during the development of configurable hardware design is critical. Suggestions are provided to decrease the reuse and porting effort and requirements are included for the development of configurable hardware design to use the platform specified abstraction.

The STRS software architecture is the focus of the current version of the STRS Standard. The software architectural model describes the relationship between the software elements, defined in layers, in an STRS-compliant radio. The model illustrates the different software elements used in the software execution and defines the API layers between an STRS application and the operating environment (OE), and between the OE and the hardware platform.

The STRS software layers are separated to enable developers to implement the software layers differently according to their requirements while still complying with the STRS architecture. A key aspect is the abstraction of the STRS application, which is either a waveform or service, from the underlying OE software to promote portability of the STRS application. The STRS software architecture uses three primary interfaces, as follows: (1) The STRS API; (2) The STRS hardware abstraction layer (HAL) specification; and (3) The Portable Operating System Interface (POSIX). The STRS API provides the interfaces that allow applications to be instantiated and use platform services. These APIs also enable communication between STRS applications and the STRS infrastructure. The HAL provides a software view of the specialized hardware by abstracting the physical hardware of interfaces. It is to be published so that software and configurable hardware design running on the platform's specialized hardware can integrate with the STRS infrastructure.

1.2.3 Roles and Responsibilities

The final configuration of an SDR and its applications is generally a product of multiple organizations performing various roles. As figure 1, Roles and Responsibilities, illustrates, the effort begins with a mission need for a radio, which could support communications, navigation, and in some instances even networking functions. The mission system engineer defines requirements. For each mission, the STRS integrators, STRS platform providers, and STRS application developers are selected. Eventually the platform and applications are integrated into the STRS compliant radio product. Both the hardware and software are tailored to meet mission-specific needs.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

The STRS platform provider is the organization responsible for the design and development of the SDR hardware platform, including the STRS OE (e.g. infrastructure, OS), configuration files, XML schema, etc. and associated documentation. The OE and hardware platform are a unique set and become the SDR platform.

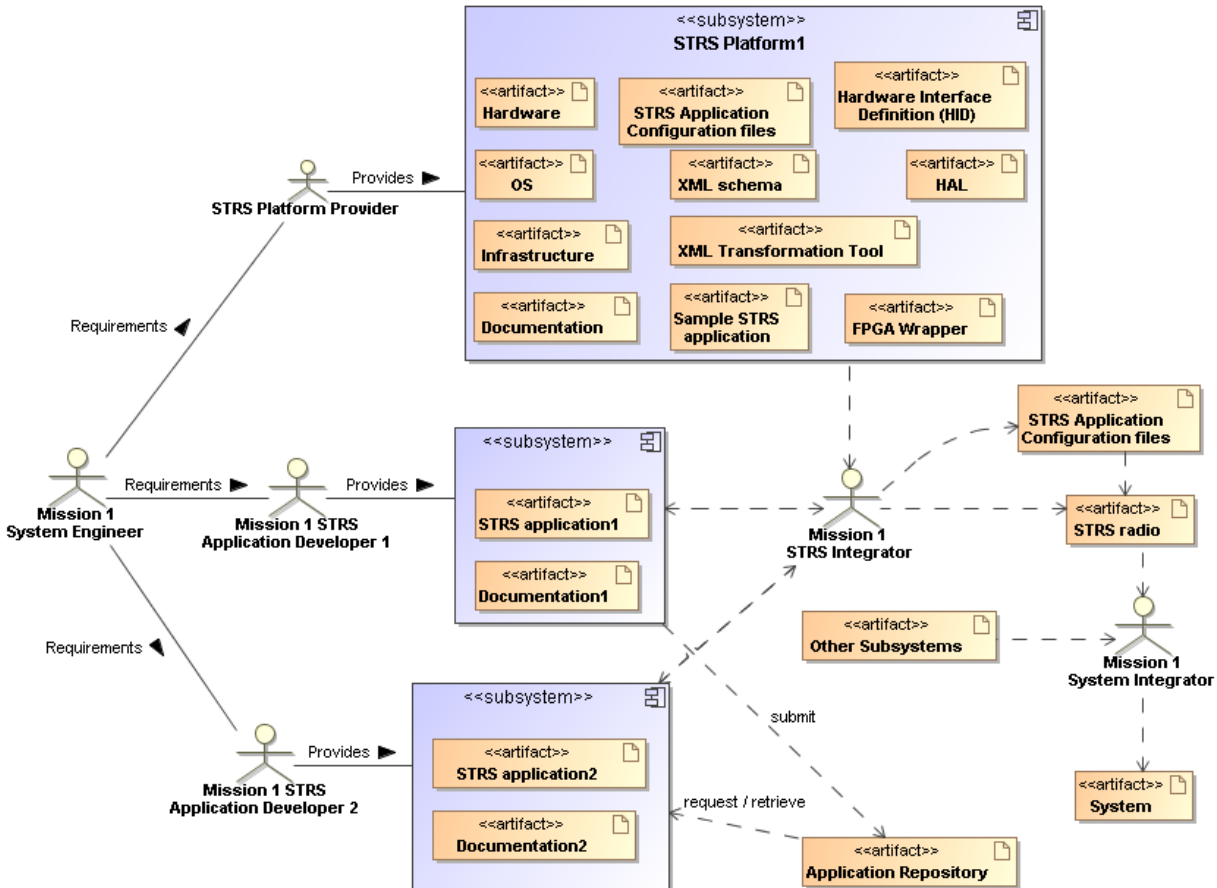


Figure 1—Roles and Responsibilities

The STRS platform provider is responsible for all the documentation associated with the platform including the user's guide, development guides, documentation, the HAL, and HID. The STRS platform provider is responsible for the FPGA platform-specific wrapper and software header files specifying the required interface, constants, typedefs, and structs. The STRS platform provider is also responsible for the STRS configuration file formats, XML schema, and transformation tool. If the STRS platform provider delegates responsibility for part of the OE to a separate infrastructure provider, the responsibility for the appropriate files and documentation may be delegated to that provider as well. If the STRS platform provider delegates responsibility for part of the hardware to a separate hardware provider, the responsibility for the pertinent HID documentation may be delegated to that hardware provider as well. The STRS platform provider is ultimately responsible to integrate and deliver all aspects of the platform and OE documentation.

The mission and the STRS application developer have the responsibility to evaluate the contents of the STRS repository against the mission-developed application requirements and determine if a new application should be developed or if an appropriate application exists in the repository that is a candidate for a port to the defined platform. Depending on the results of this decision, the STRS application developer either creates a new application or ports an existing STRS application, usually retrieved from the STRS repository. The STRS application developer performs unit tests, and documents the functionality.

The STRS integrator brings the hardware platform and software application together on the SDR platform. The STRS integrator could be the STRS platform provider, the STRS application developer(s), a mission engineer, or even a third party. The STRS integrator's role is to have the application properly running on the SDR platform to meet the communication, navigation, or other functions of the mission. Once the STRS radio integration is complete, it is delivered to a system integrator who incorporates it into the mission spacecraft system. Software updates are possible during the STRS radio and system integration. Following system integration, the STRS application developer delivers the version of the software used for the deployed system, and the associated documentation, to the STRS repository.

It is likely that multiple applications will be developed for a single STRS platform, prior to deployment and during its operational lifetime. During operations, after the radio has been deployed, additional application providers, who may be independent of the original platform or application provider, could develop additional applications for the original STRS radio. The new providers develop applications for the SDR platform much like the original application provider and deliver the application to the same or possibly a different STRS integrator. Following successful integration, the application software is delivered to the STRS application repository. Mission operations performs the role of system integrator when uploading the application to the STRS radio.

For the next mission (mission 2), either a derivative of the initial platform or a new STRS-compliant platform is envisioned. The mission 2 application provider may withdraw applications from the repository to use for the new STRS radio project. The mission 2 application follows a similar path of delivery to the mission 2 STRS integrator who incorporates the new hardware platform material and delivers the mission 2 STRS radio based on the original application and new hardware platform. As more and more missions deploy SDRs, new platforms and applications may be developed but also platforms and software are reused, marking the significant difference with the new technology compared to legacy radios.

1.2.4 Background

The deployment of SDRs for NASA missions was a new concept in 2002 due to the development of reconfigurable components useable for space radios. The need to reduce the cost and risk of using SDRs was identified and the development of the STRS architecture was initiated. In 2007, the architecture was determined to be ready for flight implementation in a technology development project. This project was originally called the Communication, Navigation, and Networking reConfigurable Testbed (CoNNeCT). CoNNeCT was later renamed the SCA_N Testbed. Three SDRs, compliant with the STRS architecture, were procured in 2008 and 2009

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

for the SCaN Testbed, using the architecture defined in a technical memorandum and referred to in the procurement specifications as Version 1.02.1. The SCaN Testbed was launched in July 2012 and operates on an external truss on the International Space Station (ISS).

The SCaN Testbed is an experimental communications system that provides the capability for S-Band, Ka-Band, and L-Band communication with space and ground assets. Investigation of SDR technology and the STRS architecture are the primary focus of the SCaN Testbed. As a completely reconfigurable testbed, the SCaN Testbed provides experimenters an opportunity to develop and demonstrate experimental waveforms and applications for communication, networking, and navigation concepts and to advance the understanding of operating SDRs in space. Lessons learned from the STRS platform provider, STRS application developers, and STRS integrators of the SCaN Testbed provided critical insight for the development of the current Standard contained in this document. The updates from the Version 1.02.1 Technical Memorandum to the NASA-STD-4009 can be requested from the STRS project.

1.3 Applicability

This Standard is applicable to space and ground SDRs developed by or for NASA missions.

This Standard is approved for use by NASA Headquarters and NASA Centers, including Component Facilities and Technical and Service Support Centers, and may be cited in contract, program, and other Agency documents as a technical requirement. This Standard may also apply to the Jet Propulsion Laboratory or to other contractors, grant recipients, or parties to agreements only to the extent specified or referenced in their contracts, grants, or agreements.

Requirements are numbered in the form (STRS-###) and indicated by the word “shall.” Explanatory or guidance text is indicated in italics beginning in section 4.

1.4 Tailoring

Tailoring of this Standard for application to a specific program or project shall be formally documented as part of program or project requirements and approved by the Technical Authority.

2. APPLICABLE DOCUMENTS

2.1 General

The documents listed in this section contain provisions that constitute requirements of this Standard as cited in the text.

2.1.1 The latest issuances of cited documents shall apply unless specific versions are designated.

2.1.2 Non-use of specific versions as designated shall be approved by the responsible Technical Authority.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

The applicable documents are accessible via the NASA Standards and Technical Assistance Resource Tool at <https://standards.nasa.gov> or may be obtained directly from the Standards Developing Organizations or other document distributors.

2.2 Government Documents

None.

2.3 Non-Government Documents

Institute of Electrical and Electronics Engineers (IEEE)

Note: The following document is the current version of the POSIX™ standard as of 2003 applicable to the requirement STRS-90.

Document Number	Document Title
IEEE 1003.13™-2003	IEEE Standard for Information Technology—Standardized Application Environment Profile (AEP)—POSIX® Realtime and Embedded Application Support

2.4 Order of Precedence

This Standard establishes requirements for an architecture standard for NASA space communication radio transceivers but does not supersede nor waive established Agency requirements found in other documentation.

2.4.1 Conflicts between this Standard and other requirements documents shall be resolved by the responsible Technical Authority.

3. ACRONYMS AND DEFINITIONS

3.1 Acronyms and Abbreviations

ADC	analog-to-digital converter
AEP	application environment profile
AGC	automatic gain control
ANSI	American National Standards Institute
API	application program interface
APP	application
ASCII	American Standard Code for Information Interchange
ASIC	application-specific integrated circuit
BIT	built-in test
BSP	board support package

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

C++	computer programming language
C&DH	command and data handling
CCSDS	Consultative Committee for Space Data Systems
CoNNeCT	Communication, Navigation, and Networking reConfigurable Testbed (This name has been replaced with ScaN.)
COTS	commercial off the shelf
DAC	digital-to-analog converter
DC	direct current
DEC VMS	Digital Equipment Corporation Virtual Memory System
DLL	dynamic link library
DSP	digital signal processor
EDIF	electronic design interchange format
EEPROM	electrically erasable, programmable read-only memory
FIFO	first in, first out
FIPS PUB	Federal Information Processing Standard Publication
FPGA	field programmable gate array
GPIO	general-purpose input output
GPM	general-purpose processing module
GPP	general purpose processor
GPS	global positioning system
HAL	hardware abstraction layer
HDL	hardware description language
HID	hardware interface description
HW	hardware
I/O	input/output
ID	identification, identifier
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronic Engineers
IF	intermediate frequency
INCITS	InterNational Committee for Information Technology Standards
IP	internet protocol
ISO	International Standards Organization
ISS	International Space Station
JTC	Joint Technical Committee
JTRS	Joint Tactical Radio System
LLC	logical link control
LNA	low-noise amplifier
LRU	logical replaceable unit
MAC	medium access control, a sublayer of the open system interconnection data link layer
MDA	model-driven architecture

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

mVpp	millivolt peak-to-peak voltage
MMU	memory management unit
NASA	National Aeronautics Space Administration
NM	network module
NPR	NASA Procedural Requirement
OAL	OEM adaptation layer
OE	operating environment
OEM	original equipment manufacturer
OM	optical module
OMG	Object Management Group
OTAP	over-the-air programming
ORMSC	Operational Research MSc Programmes
OS	operating system
OSS	open source software
PIM	platform-independent model
POSIX	Portable Operating System Interface
PROM	programmable read-only memory
PSE51	minimal realtime system profile, defined in IEEE Std 1003.13
PSE52	realtime controller system profile, defined in IEEE Std 1003.13
PSE53	dedicated realtime controller system profile, defined in IEEE Std 1003.13
PSE54	multi-purpose realtime system profile, defined in IEEE Std 1003.13
PSM	platform-specific model
PUB	publication
RAM	random access memory
RF	radio frequency
RFM	radio frequency module
ROM	read-only memory
RPN	reverse Polish notation
RT	reconfigurable transceiver
RTOS	real-time operating system
SCA	Software Communications Architecture
ScaN	Space Communications and Navigation (new name for CoNNeCT)
SDR	software-defined radio
SEC	security module
SEU	single-event upset
SPM	signal-processing module
SRAM	static random access memory
STRS	Space Telecommunications Radio System

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

SWRADIO	software radio
TT&C	telemetry, tracking, and command
TCP	transmission control protocol
TM	technical memorandum
TMR	triple-mode redundancy
TP	technical publication
UML	Unified Modeling Language
UNIX	computer operating system developed by AT&T Bell Laboratories.
VHDL	VHSIC hardware description language
VHSIC	very-high-speed integrated circuit
VMS	Virtual Memory System
Windows NT	Windows operating system—NT, new technology
XML	Extensible Markup Language
XPath	XML Path Language
XSD	XML 1.0 Schema Definition
XSL	Extensible Stylesheet Language
XSLT	Extensible Stylesheet Language Transformation

3.2 Definitions

To improve the understanding of material presented in the STRS documents, new terms and definitions that are rapidly emerging in the field of SDRs are provided below, as follows:

Adaptability: Ease with which a system satisfies differing system constraints and user needs.

Application: Executable software program that exhibits predetermined functionality and may contain one or more software modules.

Note: A primary example of an STRS application is the waveform application. An STRS application is to comply with the architecture.

Application Program Interface (API): Formalized set of software calls and routines that can be referenced by the application program in order to access supporting system or network services.

Architecture: Organizational structure of a system, the relationships between its components, and the principles and guidelines governing their design and evolution over time.

Autonomous Operation: Implementation decision-making algorithm that can be implemented on a system level (fully autonomous) or at the subsystem level (semi-autonomous) according to mission requirements.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Availability: Degree to which a system or component is operational and accessible when required for use.

Board Support Package (BSP): Hardware abstraction of the general purpose processing module (GPM) for the POSIX-compliant operating system (OS), which contains the boot, generic and processor-specific drivers required for the specific hardware.

Note: The BSP leverages commercial-off-the-shelf (COTS) device drivers and other software necessary for applications to access the specific hardware.

Built-In Test: Internal test to determine whether or not the STRS radio and each subsystem are working properly.

Note: STRS health management uses BIT to automatically monitor the health of the system and to pass any identified problem to the fault management. STRS fault management uses BIT to automatically monitor, diagnose, and isolate system problems.

Common Platform: Generic set of hardware and software radio modules that meets the requirements for multiple mission types.

Component: Hardware or software that make up a system, which may be subdivided into other parts or units.

Note: The terms “module,” “component,” and “unit” are often used interchangeably or defined to be subelements of one another in different ways depending upon the context. The relationship of these terms is not yet standardized.

Configurable Hardware Design: The electronic files used to configure the portion of the SDR hardware that can be updated after deployment.

Note: Configurable hardware design is often informally – and often incorrectly - referred to as firmware. The term firmware is defined by the IEEE Standard Glossary of Software Engineering Terminology, Std 610.12-1990, as follows: “The combination of a hardware device and computer instructions and data that reside as read-only software on that device. Notes: (1) This term is sometimes used to refer only to the hardware device or only to the computer instructions or data, but these meanings are deprecated. (2) The confusion surrounding this term has led some to suggest that it be avoided altogether.”

For this Standard, to avoid confusion the term “firmware” is not being used. The term “configurable hardware design” was selected instead. For a configurable hardware device, such as an FPGA, it includes the FPGA source

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

code written in HDL, the image stored in random access memory (RAM) and used by the FPGA, and supporting configuration files, if applicable.

Data Publisher: Software component that transmits data to one or more subscribers.

Note: In the STRS architecture, it may be implemented by waveforms and parts of the STRS infrastructure.

Data Subscriber: Software component that receives data from the data publisher.

Note: In the STRS architecture, it may be implemented by waveforms and parts of the STRS infrastructure.

Deployment: All the processes involved in getting new software or hardware up and running properly in its environment, including installation, configuration, running, testing, and making necessary changes.

Evolvability: Ease with which a system or component can be modified to take advantage of new software or hardware technologies.

External Interface: Functional and physical connections at the boundaries of a system that are designed to interoperate with other systems or components.

Note: Examples include interfaces to or from the flight computer, power, data sources, data sinks, antennas, mounting locations, and optical links.

Fault Management: Set of functions that detect, isolate, and correct malfunctions within the system or provide notifications.

Flexibility: Ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed.

Flight Computer: Separate computer that is used to monitor and control the STRS radio.

Note: The flight computer may be connected to the STRS radio electrically, electromagnetically, optically, etc. The flight computer may contain the watchdog timer for the STRS radio.

General-Purpose Processing Module: Hardware module that contains and executes the STRS OE and STRS applications and services software.

Note: The GPM consists of the general purpose processor (GPP), appropriate memory (both volatile and nonvolatile), system bus, the spacecraft (or host) TT&C interface, ground support telemetry and test interface, and the components to support the radio configuration.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Guidelines: Nonbinding statements intended to direct the broader and longer-term aspects of the STRS architecture.

Hardware Abstraction Layer: Library of functions that provides a software view of the specialized hardware by abstracting the physical hardware interfaces.

Hardware Device: Physical entity that is capable of performing a function.

Hardware Interface Description: Documentation containing information about each module's physical and electrical connections, performance, capability, size, weight and power, as applicable, to enable integration between components of the system.

Health Management: Monitoring the health and performance of a system, subsystem, device, or process.

Note: Health management invokes fault management, when corrective action is needed.

Hierarchical Structure: Structure that characterizes a system in which components are contained by other components and/or provide services to the next higher-level components.

Note: Hierarchical structure is a key attribute of an open architecture that enables system description, design, development, installation, operation, upgrades, and maintenance to be performed at a given layer or layers. This type of structure allows each layer to be modified without affecting the other layers.

Interoperability: (1) Ability of a system to work with or use the parts or equipment of another system; (2) capability of different radio systems or radio networks to communicate and exchange information with each other.

Note: Dissimilar systems or networks may achieve interoperability by changing their operating parameters to a common compatible format or by operating through a bridge that translates between incompatible formats. An alternate definition is to determine and adapt all radio parameters required for broadest communication compatibility across all target networks.

Legacy Radio: Nonprogrammable radio designed for one fixed configuration that produces a single waveform at a specified frequency.

Note: The radio may have limited options for tuning, data rate, and so forth or may even carry multiple types of data, but it is incapable of adapting to new waveforms.

Maintainability: Ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Method: Implementation of an operation, which specifies the algorithm or procedure associated with an operation.

Module: Self-contained hardware or software component that interacts with a larger system.

Note: A software module (program module) performs specific tasks within a software system. A hardware module is a physical grouping of devices capable of implementing specific functions.

Open Architecture: Architecture whose functions, interfaces, components, and/or design rules are defined and published.

Open Source or Open-Source Software (OSS): Any computer software distributed under a license that allows users to change and share the software freely.

Note: OSS is required to have its source code freely available, and end-users have the right to modify and redistribute the software to others.

Open System: System that has specified, publicly maintained, and readily available standards.

Over-the-Air Programming (OTAP): Method of providing software updates by means of a communication channel realized by the STRS radio itself.

Portability: Ease with which a system application or service can be transferred from one hardware or software environment to another.

Portable Operating System Interface: Family of IEEE standards 1003.n that describes the fundamental operating system (OS) services and functions necessary to provide a UNIX-like kernel interface to applications.

Note: POSIX is not an OS but ensures that programming interfaces are available to the application programmer.

Queue: List in which items are appended to the last position in the list and retrieved from the first position in the list; that is, the next item to be retrieved is the item that has been in the list for the longest time.

Radio Frequency Module (RFM): Module that converts to and from carrier frequencies and provides the signal-processing module with baseband or intermediate frequency (IF) signals and provides the transmission and reception equipment with radio frequency (RF) signals.

Note: RFM-associated components may include filters, RF switches, duplexers, low noise amplifiers (LNAs), power amplifiers, analog-to-digital (ADC) converters, and digital-to-analog (DAC) converters. This module handles the interfaces that

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

control the final stage of the transmission or first stage of the reception of the wireless signals, including antennas.

Real-Time Operating System (RTOS): OS that guarantees a certain capability within a specified time constraint.

Reconfigurability: Ability to modify functionality of a radio by changing the operational parameters without requiring a software update.

Reconfigurable Radio: Radio whose functionality can be changed either through manual reconfiguration of radio modules or under software control.

Note: Software reconfiguration control of such radios may involve any element of the radio-communication network. SDRs are a subset of reconfigurable radios.

Reconfigurable Transceiver (RT): Radio with limited processing and selectable remote reconfiguration (e.g., filter parameters and modulations).

Reconfiguration: Act of modifying the functionality of a radio by changing the operational parameters without updating the software.

Reentrant Function: Function that can be entered before completion of a prior execution of that same function and execute correctly.

Note: A function that is reentrant is automatically thread-safe, but not necessarily the reverse.

Reliability: Ability of a system or component to perform a required function under stated conditions for a specified period of time.

Reprogrammability: Ability to modify functionality of a radio by changing the operational software or configurable hardware design either wholly or partially.

Reusability: Degree to which a software module or other work product can be used in more than one computing program or software system.

Scalability: Degree to which components or functions in an implementation can be sized in systematic proportions for varying capacities.

Selectable: Ability to choose from a range of choices.

Note: For example, a selectable parameter may be modified to change system characteristics at runtime.

Services: Software programs that provide functionality available for use by other applications.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Signal-Processing Module (SPM): Module that contains the implementations of the signal processing used to handle the transformation of received digitally formatted signals into data packets and/or the conversion of data packets into digitally formatted signals to be transmitted.

Note: The SPM may include the spacecraft data interface, application specific integrated circuits (ASICs), FPGAs, digital signal processors (DSPs), memory, and connection fabric or bus.

Software: Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system.

Note: In certain contexts in this Standard, the term “software” also encompasses configurable hardware design. For example, in the term “software defined radio,” the word “software” includes configurable hardware design. In other contexts, the word “software” is meant to imply code running on a processor, especially in the Software Architecture section. In this case, even if the processor is embedded within configurable hardware, the software that executes on the processor is not “configurable hardware design.”

Software-Defined Radio: Radio in which some or all of the physical layer functions are implemented in software and/or configurable hardware design.

Software-Defined Radio Architecture: Comprehensive, consistent set of functions, components, and design rules according to which radio communications systems may be organized, designed, constructed, deployed, operated, and evolved over time.

Note: A useful architecture partitions functions and components such that (1) functions are assigned to components clearly, and (2) physical interfaces among components correspond to logical interfaces among functions.

Software Device: A software abstraction of a hardware device or group (aggregate) of hardware devices.

Note: An STRS device is a software device that is part of the STRS infrastructure, having a well-defined and portable API that may use the HAL to read, write, and control hardware devices.

Software Radio: Extension of an SDR with more functionality implemented in GPPs as opposed to ASICs and FPGAs. A software radio implements communications functions primarily through software in conjunction with minimal hardware.

Note: Software radios are the ideal SDR in which digitization occurs at the antenna.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Space Telecommunications Radio System: Project that defines and maintains the SDR architecture for NASA.

Specialized Hardware: Separate hardware that can be initialized or controlled using software.

Standards: Technical specifications that are widely used, consensus-based, published, and maintained by recognized industry standards organizations.

STRS Command: Source that abstracts the command functionality usually found in the interface to the flight computer.

STRS Infrastructure: Part of the STRS OE that configures and controls STRS applications and services as well as specialized hardware via the HAL.

Note: Additional functionality may be required for radio robustness and mission-dependent requirements.

STRS Operating Environment: Portion of the STRS radio that contains the STRS Infrastructure, the POSIX-conformant RTOS, the HAL, and optional middleware software.

Note: The STRS OE executes STRS services and waveform applications.

STRS Platform: Combination of hardware and software components, including the STRS OE, capable of executing software applications.

STRS Radio: SDR that is compliant with this Standard and that runs one or more waveforms.

System: Collection of components organized to accomplish a specific function or a set of functions.

System Architecture: Abstract description of the entities of a system, and the relationship between the entities.

Thread-Safe: Function that works correctly during simultaneous execution by multiple threads, without unwanted interaction between the threads. A thread is a part of a program that can execute independently of other parts. A thread is the smallest sequence of programmed instructions that can be managed independently by an OS scheduler.

Note: A function that is reentrant is automatically thread-safe, but the reverse is not necessarily true.

Upgradeability: Ability to make changes to a portion of the system easier by limiting the changes, as much as possible, to the updated part.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

Note: It is clear that greater upgradeability is greater ability.

Usability: Ease with which a user can learn to operate, prepare input for, and interpret the output of a system or component.

Use Cases: Situations that capture the requirements of a system by describing how the system should interact with the users or other systems (the actors) to achieve specific goals.

Watchdog Timer: Software and/or hardware that monitor the health of a system and, if a problem is detected, take the appropriate action to restore the system back to health.

Waveform: Set of transformations applied to information (e.g., voice or data) that is transmitted over the air and the corresponding set of transformations to convert received signals back to their information contents.

Note: Traditionally, a waveform was simply an electromagnetic signal whose amplitude varies with time.

Waveform Application: Code that implements all the functions and algorithms necessary to realize a waveform.

Note: The waveform application can be distributed among various processing elements, including specialized hardware (e.g., FPGAs and DSPs). In STRS, if the waveform application requires run-time support for functions that it cannot provide directly, it is to use the STRS APIs in the infrastructure to access the desired functions whether or not they are provided by the infrastructure directly or by other waveforms or services.

4. HARDWARE ARCHITECTURE

In addition to providing benefits by defining a standard software infrastructure for NASA's radios, this Standard also defines standards for the hardware portion of the radio. Hardware technologies usually change more rapidly than software, and each radio implementation generally has very specific spacecraft dependencies and requirements. Therefore, the STRS hardware architecture is specified at a functional level, rather than at the physical implementation level. Also, a functional-level architecture will remain applicable over a longer time frame. It should be noted that programs have the latitude to standardize hardware requirements at the implementation level for multiple radio procurements.

The STRS hardware architecture was developed with consideration of several key constraints and conditions for operating space SDRs. One major issue driving the hardware architecture formulation was the need for flexibility, so that a single architecture is capable of addressing the range of different mission classes. The mission classes have radio requirements that range from requiring small radios that are highly optimized to meet severe size, weight, and power constraints, to missions requiring complex radios with multiple operating frequencies and higher data rates. This requires that the hardware architecture accommodate a range of reconfigurable processing technologies including GPPs, DSP, FPGAs, and ASICs with selectable parameters.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

Currently, reconfigurable signal processing is primarily performed in specialized signal-processing hardware for the frequencies and data rates used in NASA space missions, and this is expected to continue for some time. In addition to providing capability, specialized signal processing is generally more power efficient than general purpose processing. Likewise, the use of FPGA-based specialized signal processors is generally more power efficient than DSP-based signal processing. The needs for specialized processing are supplemented by the software infrastructure, which is more suited for execution in a GPP. The architecture also enables technology infusion over time, accommodating the rapidly evolving capabilities of processor speeds and signal processing. In addition, the conversion point, where the signal is digitized, is moving closer to the antenna. Considering these points, the architecture provides a flexible framework, emphasizing common terminology for hardware functions and interfaces, common documentation, and common formats and requirements for waveform and service STRS application developers to utilize a platform's capabilities. The architecture does not prescribe a specific hardware implementation approach.

An STRS platform is to be delivered with a complete HID, which is described in section 5.4. The HID specifies the electrical interfaces, connector requirements, and physical requirements for the delivered radio. Each module's HID abstracts and defines the module functionality and performance.

(STRS-1) An STRS platform shall have a known state after completion of the power-up process.

4.1 Generalized Hardware Architecture and Specification

Figure 2, Hardware Architecture Diagram Key, illustrates the symbols and terminology used within the hardware architecture diagrams. The hardware diagram illustrates the radio functions and the interconnects for each module. The modules are a logical and functional division of common radio functions that comprise an STRS platform. Modules are not intended to represent physical entities of the platform. As developers choose how to distribute and implement the radio functions among hardware elements, the specification provides the guidance on the interfaces and abstractions that are to be provided to comply with the architecture. The module and function connections provided in the diagrams are data path, control, signal clock, and external interfaces.

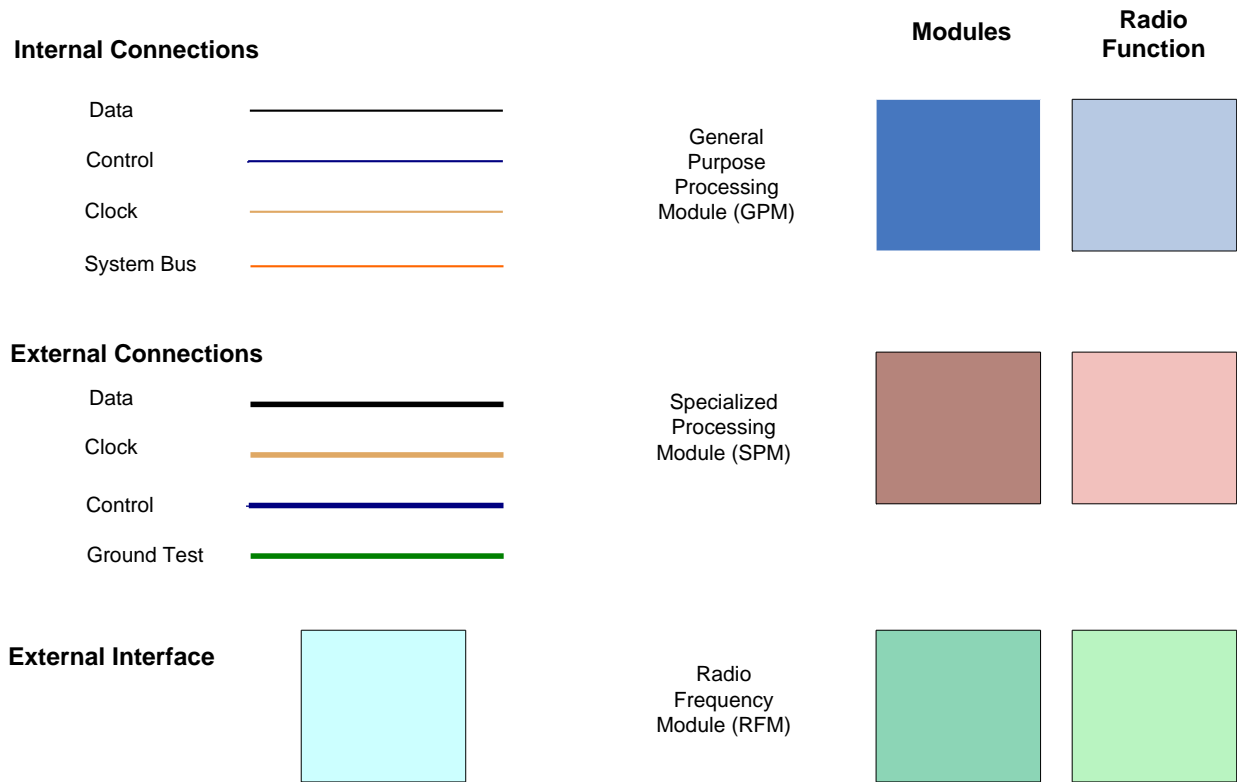


Figure 2—Hardware Architecture Diagram Key

Figure 3, *Notional STRS Hardware Architecture Implementation*, shows the high-level STRS hardware architecture. The figure illustrates the functional attributes and interfaces for each module. A module is a combination of logical and functional representations of platform and applications implemented in a radio. The modules are divided into their typical functions to provide a common description and terminology reference. Each STRS platform provider has the flexibility to combine these modules and their functionality as necessary during the radio design process to meet the specific mission requirements. Additional modules can be added for increased capability.

The hardware architecture does not specify a physical implementation internally on each module, nor does it mandate the standards or ratings of the hardware used to construct the radios. Thus the radio supplier can encapsulate company proprietary circuit or software designs, provided the modules meet the specific architecture rules and expose the interfaces defined for each module. There is flexibility to physically combine these modules as necessary during the radio design process to meet the specific mission requirements. For example, all RF and signal-processing components or functions may be integrated onto a single printed circuit board, easing footprint, interface, and integration issues, or an approach with multiple boards and enclosures could be used.

Each mission, or class of missions, may choose to standardize certain interfaces and physical packaging. This approach provides NASA with the flexibility to adopt different implementation standards for various mission classes. Thus, if a series of radios are required with common operating requirements, physical construction details, such as bus chassis or card slice, can be part of the acquisition strategy, for cost-effective modularity at a lower level to match the life cycle of the hardware. Another example of the flexibility is where a large mission class program may choose to standardize the details of the RF-to-signal-processing interface. This might be done to facilitate the use of different RF modules, but the same signal process module, for radios used for several similar missions.

Figure 3 depicts radio functions, or elements, expected for each module in a notional sense. It should be noted that not all the elements shown in each module are necessarily required for implementation. This architecture specifies the functionality of each module, but it does not necessarily specify how they are implemented. Mission requirements will dictate the implementation approach to each module, and the modules required in each radio.

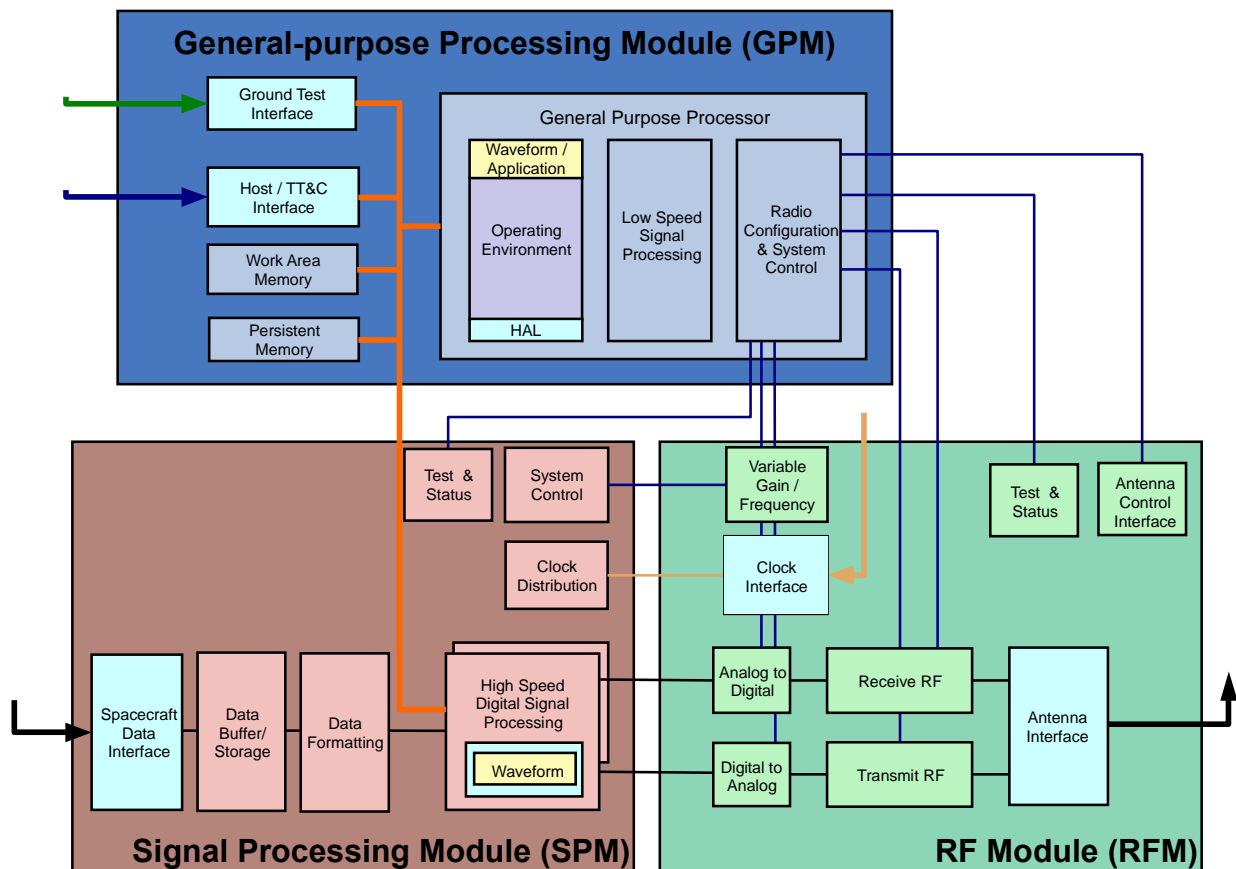


Figure 3—Notional STRS Hardware Architecture Implementation

4.1.1 Components

The approach taken in the STRS is to describe the radio hardware architecture in a modular fashion. The generic hardware architecture diagram identifies three main functional components or modules of the STRS radio. Although not shown in figure 3, additional modules (e.g., optical, networking, and security) can be added for increased capability and will be included in the specification as it matures. The hardware architecture currently consists of the following modules:

General-Purpose Processing Module: Consists of the GPP, appropriate memory (both volatile and nonvolatile), system bus, the spacecraft (or host) TT&C interface, ground test interface, and the components to support the radio configuration.

Signal-Processing Module: This module contains the implementations of the signal processing used to handle the transformation of received digitally formatted signals into data packets and/or the conversion of data packets into digitally formatted signals to be transmitted. Also included is the spacecraft data interface. Components include ASICs, FPGAs, DSPs, memory, and connection fabric or bus.

Radio Frequency Module: This module handles the RF functionality to provide the SPM with the filtered, amplified, and digitally formatted signal. For transmission it formats, filters, and amplifies the output signal. Its associated components include filters, RF switches, diplexer, LNAs, power amplifiers, ADCs, and DACs. This module handles the interfaces that control the final stage of transmission or the first stage of reception of the wireless signals, including antennas.

Security Module (SEC): Though not directly identified in the generic hardware diagram, an SEC is also being proposed to allow STRS radios to support future security requirements. The details of this module will be defined in later revisions of the architecture.

Network Module (NM): The architecture supports Consultative Committee for Space Data Systems (CCSDS) and Internet Protocols (IPs) and networking functions. However, the Network Module (NM) may be realized as a combination of both the GPM and SPM.

Optical Module (OM): This module supports the integration of optical equipment when used. The detail of this module will be defined in later revisions of the architecture. (It has many similarities to RFM, but these are for optical carriers.)

4.1.2 Functions

Test and status, fault monitoring and recovery, and radio and TT&C data-handling functions are to be implemented on all modules to some level. The details are mission specific and are stated as part of the radio acquisition. The related control and interface requirements for the shared module functions are stated in the corresponding module section.

Test and Status: Each module (or combination of modules) should provide a means to query the current health of the module and run diagnostics.

Fault Monitoring and Recovery: Each module (or combination of modules) should incorporate detection of operational errors, upsets, and major component failures. These may be caused by the radiation environment (e.g., single-event upsets (SEUs)), temperature fluctuations, or power supply anomalies. In addition to detection, mitigation, and fail-safe techniques should be employed. Each module should have a default power-up mode to provide the minimal functionality required by the mission. This fail-safe mode should have minimal software and/or configurable hardware design dependency.

Radio Data Path: SDRs can be implemented with or without the GPM in the data path. The STRS architecture supports the separation of the RFM and SPM data paths from the GPM. Giving the GPM access to the data path as an optional capability rather than a required capability allows for a more efficient implementation for medium and small mission classes and improves the overall performance for near-term implementations. If space-qualified GPM components mature with the performance capabilities required for signal processing, the GPM can exist within the data path and take on more signal-processing functionality, increasing flexibility.

STRS Radio Startup Process: The startup of the STRS infrastructure is expected to be initiated by the STRS platform boot process, so that it can receive and send external commands and instantiate applications. In order to control an STRS platform at power-up and to recover from error conditions, an STRS platform is to have a known power-up condition that sets the state of all modules. To support upgrades to the OE, an STRS platform requires the ability to alter the state (boot parameters) and/or select a boot image. The exact mechanisms and procedures used will be platform and mission specific but need to be sufficient to support upgrades to OE components, such as the OS, BSP, and STRS infrastructure.

4.1.3 Interfaces

4.1.3.1 External Interfaces

There are several key external interfaces in this architecture:

- *Host TT&C.*
- *Ground Test.*
- *Data.*
- *Clock.*
- *Antenna.*
- *Direct current (DC) power.*
- *Thermal.*

The host TT&C interface represents the typically low-latency, low-rate interface for the spacecraft (or other host) to communicate with the radio. The host telemetry typically carries all information sourced within the radio. This type of information traditionally is called the telemetry data and includes health, status, and performance parameters of the radio as well as the link in use. In addition, this telemetry often includes radiometric tracking and navigation data. The command portion of this interface contains the information that has the radio itself as the destination of the information. Configuration parameters, configuration data files, new software data files, and operational commands are the typical types of information found on this interface.

The Ground Test Interface provides a “development-level” view of the radio and is exclusively used for ground-based integration and testing functions. It typically provides low-level access to internal parameters not typically available to the Spacecraft TT&C Interface. It can also provide access when the GPM is not functioning (i.e., during boot).

The Data Interface is the primary interface for data that are sourced from the other end of the link and for data that are sunk to the other end of the link. This interface is separate from the TT&C interface because it typically has a different set of transfer parameters (protocol, speeds, volumes, etc.) than the TT&C information. A common interface point in the spacecraft for this type of interface is the spacecraft solid-state recorder rather than the spacecraft command and data-handling (C&DH) subsystem. This interface is also characterized by medium to high latency and high data rates.

The Clock Interface is used to input to the radio the frequency reference sufficient for supporting navigation and tracking. This type of input frequency reference is essential to the operation of the radio and provides references to the SPM and RFM.

The Antenna Interface is used to connect the electromagnetic signal (input or output) to the radiating element or elements of the spacecraft. It also includes the necessary capability for switching among the elements as required. Steering the elements, if a function of the overall telecommunications system, is possible through this interface, but it is not typically employed because of overall operational constraints.

The Power Interface, which is not included on the diagram, is described as part of this specification at the highest levels. The Power Interface defines the types and conditions of the input energy to power the radio.

4.1.3.2 Networking

A networking interface does not necessarily map directly to the SPM, GPM, or RFM. The networking interface might handle only spacecraft TT&C data or both spacecraft TT&C data and radio data. This architecture allows for those capabilities.

4.1.3.3 Internal Interfaces

To support the overall goals of the architecture, the internal interfaces (GPM system bus, GPM RFM control, SPM-to-GPM test, frequency reference, and data path) should be well documented and available without restriction.

The GPM System Bus (orange lines in figure 3) provides the primary interconnect between elements of the GPM. The GPM System Bus may provide an interface between the microprocessor, the memory elements, and the external interfaces (TT&C and Test) of the GPM. The GPM System Bus is the primary interface between the GPM and the SPM, as shown in the interconnection with the major SPM processing elements. Finally, the GPM System Bus provides the interface by which the reprogrammable and reconfigurable elements of the SDR are modified. It supports both the read and write access to the SPM elements, as well as the reloading of configuration files to the FPGAs.

The interface between the GPM and the RFM is primarily a control/status interface. Various RFM elements are controlled by the set of GPM RFM control lines (blue lines in figure 3). Coming from the System Control block in the GPM, this control bus can be either fixed by the System Control function or programmed by the GPM software and validated and routed by the System Control function. It is important to have a hardware-based confirmation and limit-check on the software controlling any RFM elements. The System Control module of the GPM provides this functionality, thus keeping the GPM RFM Control bus within operational limits.

The Ground Test Interface (green line in figure 3) provides specific control and status signals from different modules or functions to the Ground Test Interface block. This interface is used during development and testing to validate the operation of the various radio functions. This interface is very specific to the implementation and realization of the different modules.

The Frequency Reference Interface provides an important interface between the RFM and the SPM functions. It ties the two modules together in a way that allows for the SDR to implement tracking and navigation functions. The characteristics of this interface are defined by the various amounts of tracking accuracy that are required for the SPM to accomplish. This interface can be as simple as a single, common frequency reference that is conditioned from an outside source and distributed in the least degrading fashion possible to the SPM.

Finally, the data paths are the various streams of bits, symbols, and RF waves connecting the major blocks of the primary data path. For any particular implementation, the data path or bitstreams are defined by the particular application implemented in the functional blocks. The interface between the RFM and SPM should be well defined and have characteristics suitable for that level of conversion between the analog and digital domains.

The hardware architecture can be further specified in a manner that is important for implementers to consider and follow, if the implementation dictates the necessity of particular components. Details of the GPM, SPM, and RFM are provided in subsequent sections.

4.2 Module Type Specification

4.2.1 General-Purpose Processing Module

Figure 4, GPM Architecture Details, provides a closeup of the GPM detail. The GPM consists of one or more general purpose or digital signal-processing elements and support hardware components, embedded OS, software applications and interfaces to support the configuration, control, and status of the radio. The number of processing elements and the extent of support hardware will vary depending on the mission-class processing and data-handling requirements from a single system on a chip implementation for smaller mission classes to multiple logical replaceable units (LRUs) for the largest mission classes. In addition, the fault tolerance requirements can also increase the number of hardware processing elements, support hardware components, and interface points required to meet the range of mission classes. The majority of processing functions of the GPM will be under software control and supported by an OS. There are cases, depending on the data-handling speeds, that require the use of separate specialized support hardware (FPGA or ASIC chips) to alleviate the burden on the processing elements. Such specialized support hardware could include encryption, packet routing, and network processing-type functions.

4.2.1.1 GPM Components

The GPM contains, as necessary, a GPP and various memory elements as shown in figure 4. Depending on the particular mission class, not all memory elements are required. The GPP will typically be implemented as a microprocessor, but it could take many forms, depending on the mission class. Because the GPM is the primary control component of the radio, it is a required module for an STRS radio. A description of each element follows.

The GPP functions include the OE, the HAL, and potentially application functions. The OE contains the STRS infrastructure, which provides the functionality for the interfaces defined by the STRS API specification. The OE also contains the OS and the POSIX abstraction layer.

The Persistent Memory Storage element holds both the permanent (e.g., programmable read-only memory (PROM)) and reprogrammable code for the GPP element. In today's technology, this code is implemented using a reprogrammable technology, such as electrically erasable, programmable read-only memory (EEPROM). It is also possible, but not typically qualifiable, to implement this code storage in Flash memory.

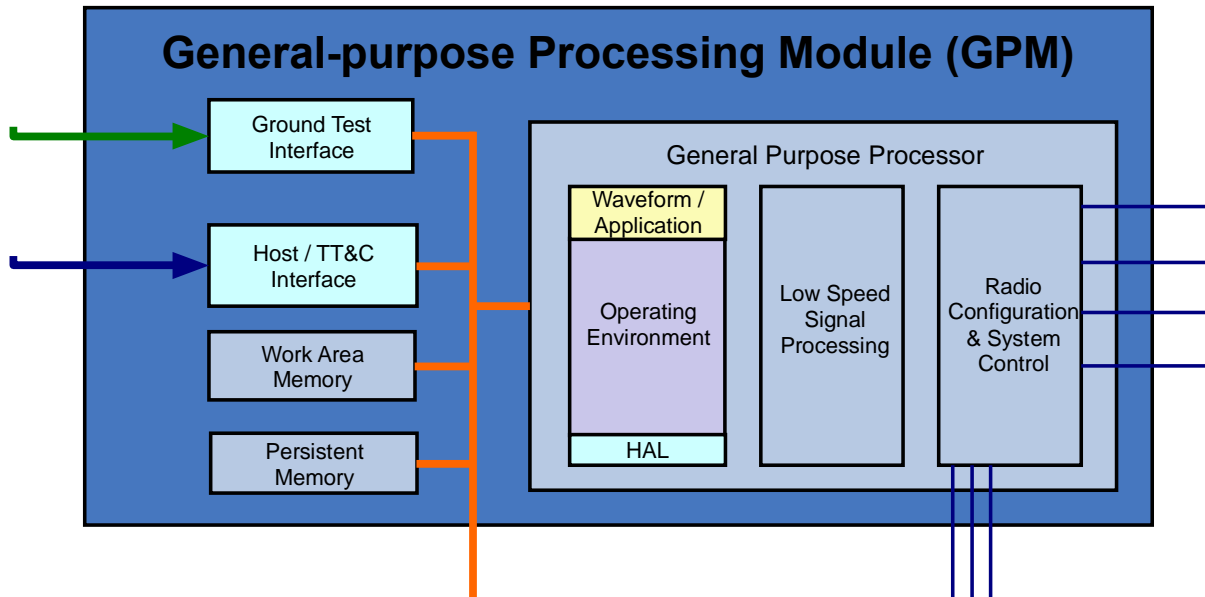


Figure 4—GPM Architecture Details

The Persistent Memory also provides the reprogrammable storage for the SPM FPGA elements (i.e., configurable hardware design). The GPM is responsible for programming and scrubbing the SPM FPGAs and therefore contains the appropriate “code” for the FPGAs. This memory block is typically implemented using a nonvolatile memory technology, such as EEPROM, but could, in particular implementations, be implemented with PROM technology.

The Work Area Memory element is provided as operational, scratch memory for the GPP element. This memory element is implemented in concert with the GPP element and may contain both data and code, as appropriate for the execution of the radio application running in the GPM.

Finally, the GPM contains a System Control element to control and moderate the GPM System Bus. This element provides the necessary control for the System Bus including the various memory and SPM elements interfaced by the System Bus. In addition, the System Control element provides a validated interface to the RFM hardware via the GPM RFM Control Interface. As the software running on the GPP element commands the RFM elements into certain states, those commands are interpreted by the System Control element and validated in a manner that will prevent damaging configurations of the RFM; for example, tying the transmit amplifier directly to the receive amplifier, bypassing the diplexer element. This level of validation in the GPM-to-RFM interfaces would prevent damage to the radio from a software bug. The System Control element is typically implemented by a non-reprogrammable (in-flight) FPGA allowing for flexibility between instantiations of a particular implementation.

4.2.1.2 GPM Functions

The GPM will provide the overall configuration and control of the STRS architecture and may include any or all of the following functions:

- a. Management and Control.*
 - (1) Module discovery.*
 - (2) Configuration control.*
 - (3) Command, control, and status.*
 - (4) Fault recovery.*
 - (5) Encryption.*
- b. STRS infrastructure, radio configuration and control.*
 - (1) Radio control.*
 - (2) System management.*
 - (3) Application upload management.*
 - (4) Device control.*
 - (5) Message center.*
- c. External network interface processing.*
- d. Internal data routing.*
- e. Waveform data link layer.*
 - (1) Medium Access Control (MAC) and Logical Link Control (LLC) layer.*
 - (2) Physical layer processing.*
- f. Onboard data switch.*

4.2.1.3 GPM Interfaces

- a. TT&C Interface.*
- b. Ground Test Interface.*
- c. Provides programmable general-purpose input output (GPIO) to support.*
 - (1) Interrupt source/sink.*
 - (2) Application data transfer.*
- d. Provides control/configuration interfaces.*
 - (1) RFM, antenna, power amplifier, and SPM.*
- e. System Bus interface.*

4.2.1.4 GPM Requirements

(STRS-2) The STRS OE shall access each module's diagnostic information via the STRS APIs.

(STRS-3) Self-diagnostic and fault-detection data shall be created for each module so that it is accessible to the STRS OE.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

(STRS-109) An STRS platform shall have a GPM that contains and executes the STRS OE and the control portions of the STRS applications and services software.

4.2.2 Signal-Processing Module

An SPM is optional for an STRS platform. The SPM may implement the signal processing used to transform received digital signals into data packets and/or the conversion of data packets into digital signals to transmit. The complexity of this module is based on the applications and data rates selected for a mission. The SPM modules contain components and capabilities to manipulate and manage digital signals that require higher processing capabilities than that supplied by the GPM. The configurable hardware design architecture describes a common interface for the application on the SPM, as described in section 6. Figure 5, SPM Architecture Details, illustrates the SPM module details.

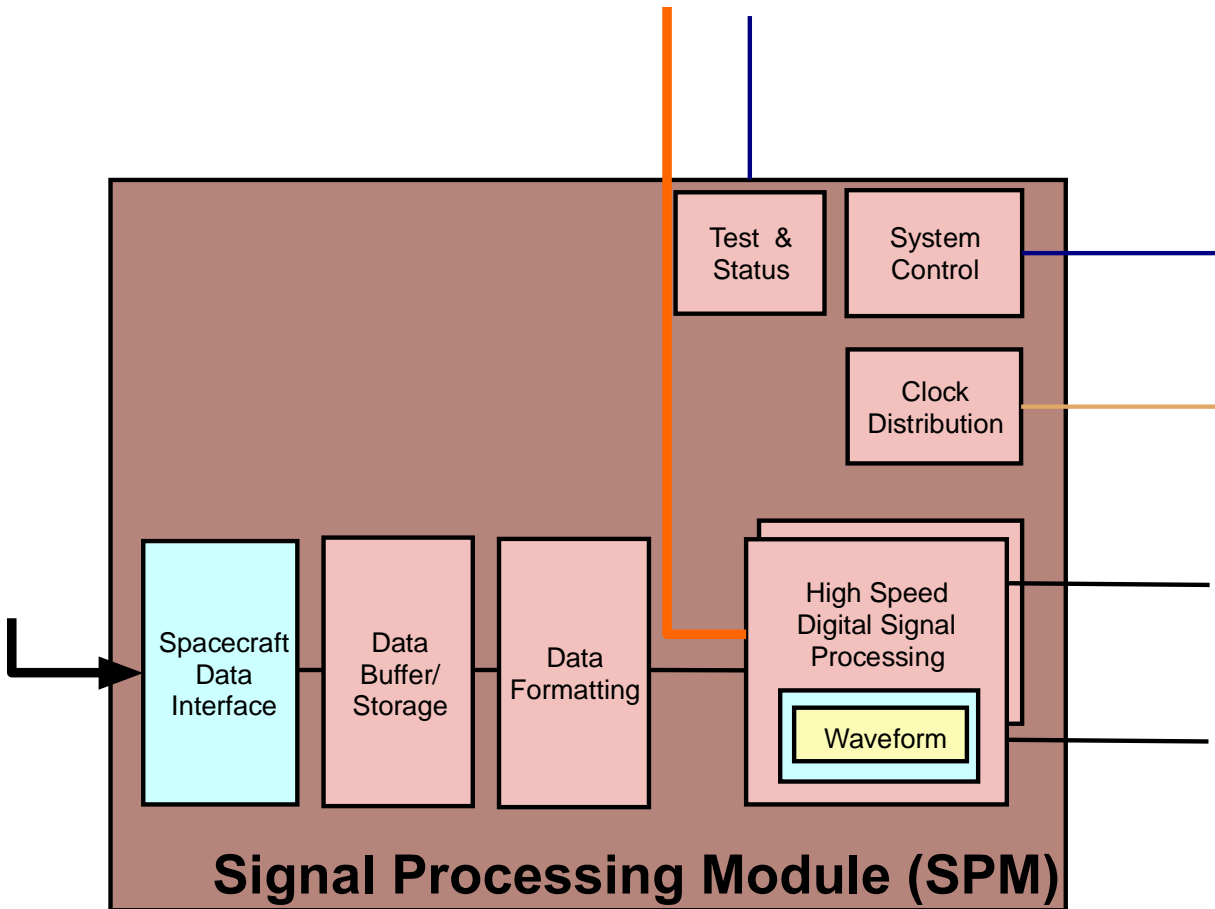


Figure 5—SPM Architecture Details

4.2.2.1 SPM Components

The SPM will initially be implemented primarily with FPGAs, DSPs, reconfigurable processors, ASICs, and other integrated circuits. However, technologies will change over time, so the specific implementation is left to the STRS platform provider.

It is also anticipated that STRS platforms may use dedicated SPM slices for specific applications and technologies. For example, a dedicated global positioning system (GPS) receiver slice can complement the existence of reconfigurable SPM slices in the same radio. The dedicated slice offloads demands on the less specific SPM. If an STRS platform contains an SPM slice, the slice should meet the module interface specifications for control and configuration and have an interface with the GPM via the GPM System Bus and the SPM-to-GPM test interface. These two interfaces work in concert to provide a control and reprogramming path to the SPM from the GPM and the application running on the GPM.

4.2.2.2 SPM Functions

The SPM performs the digital signal-processing functions, which are used to convert symbols to bits (and vice versa). These functions are typically implemented on FPGAs, DSPs, or ASICs. It is recommended that these devices be reconfigurable and reprogrammable because this allows for new applications to be implemented on the SDR in the future without a hardware redesign. However, mission-specific requirements may dictate that the application be implemented on a non-reprogrammable hardware device.

In addition to the digital signal-processing functions, a data-formatting function is typically provided to convert blocks of data stored in the data storage element into bitstreams appropriate for encoding into symbols (and vice versa). In many cases, it is possible to implement the data-formatting function in the same device as the digital signal-processing function, but that is an implementation detail dependent on the mission class.

A data storage element is used to provide a queuing buffer between the data interface and the bitstream coders and decoders. This data storage function can be implemented in either volatile or nonvolatile memory, depending on the requirements of the mission implementation.

An SPM may implement any or all of the following digital communication functions depending upon the mission waveforms:

- *Digital up conversion—interpolation, filtering, and “local oscillator” multiplication of baseband samples to obtain an IF or RF output sample stream, appropriate for digital-to-analog conversion. This is typically the last transmit function implemented in the SPM, and the output samples are sent to the RFM.*
- *Digital down conversion—multiplication with “local oscillator,” downsampling, and filtering IF or RF samples to obtain a baseband output sample stream. This is typically the first receive function implemented in the SPM, with input samples coming from the analog-to-digital conversion in the RFM.*
- *Digital filtering—averaging, low-pass, high-pass, band-pass, polyphase, and other filters used for pulse shaping, matched filter, etc. This may overlap with some of the functionality in the Up and Down Conversion.*
- *Carrier recovery and tracking—retrieval of the waveform carrier within the receive sample stream. Typical SPM functions for carrier recovery include shifting the recovered carrier frequency to compensate for local oscillator variations and Doppler shifts in the link.*
- *Synchronization (data, symbol, etc.)—alignment of received samples with symbol and data boundaries. There may be some integration with the Digital Down Conversion and Carrier Recovery and Tracking functions.*
- *Forward error correction coding—encoding transmit data so that receive data errors may be corrected to some level, enhancing the waveform performance.*
- *Digital automatic gain control (AGC)—scaling of the receive samples to optimize downstream operations.*

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

- *Symbol mapping (modulation)*—translating transmit data bits to modulation symbol samples.
- *Data detection (demodulation)*—translating receive symbol samples to data bits.
- *Spreading and despreading*—a form of encoding data to obtain certain energy dispersion in the frequency domain.
- *Scrambling and descrambling*—a form of encoding data to ensure a certain level of randomness in the digital data stream, usually for synchronization of the receiver.
- *Encryption and decryption*—a form of encoding data for security purposes.
- *Data Input/Output (I/O) (high-speed direct from or to source or sink)*—interface for transmit and/or receive data to come in or out of the module. This may require buffering and some protocol handling.

4.2.2.3 SPM Interfaces

The SPM's functions and external interfaces are shown in figure 5. Interfaces shown include those common to all module types as well as those specific for the SPM. These SPM-specific interfaces may not all be required for some missions. Note that the implementation of these interfaces may combine two or more on one physical transport. For example, the Data Interface and Control and Configuration Interfaces may both use the same physical Serial Rapid IO connection.

- *Data I/O to or from RFM*—This is the digital sample stream going to the RFM's DACs for transmission, and the digital samples from the RFM's ADCs. However, if the DACs and ADCs are preferred to be a part of the SPM, then this interface is replaced with analog baseband or IF signals.
- *Waveform control and feedback to RFM*—This interface will be waveform dependent. It may be used, for example, to send feedback to an AGC or control frequency hopping.
- *Data interface external to the radio*—High-data-rate waveforms may need a direct interface to the SPM if the GPM is not designed to handle the data.
- *System bus—Data to or from GPM*—This interface exchanges the packetized data for transmission and reception.
- *Control and configuration from GPM*—Waveform loads and reconfigurable parameters are managed through this interface.
- *Test and status to GPM*—Tests are initiated through this interface by the GPM, and results are returned. This is a more basic interface (electrically and protocol-wise) than the Control and Configuration interface.
- *Radiometric tracking.*

The HID is to contain the characteristics of each reconfigurable device. Reconfigurable capacity is usually measured by the number of FPGA gates, logic elements, or bytes. This information can

be used by future STRS application developers to determine the waveforms that can be implemented on a given platform.

4.2.3 Radio Frequency Module

The RFM handles the conversion to and from the carrier frequency, providing the SPM and/or the GPM with digital baseband or IF signals, and the transmission and reception equipment with RF to support the SPM and GPM functions. Its components typically include DACs, ADCs, RF switches, up converters, down converters, diplexer, filters, LNAs, power amplifiers, etc. Current and near-term RF technologies cannot be expected to allow multiband operation using a single channel RFM, and thus multiband radios will require the use of multiple RFM slices. The RFM provides a band of frequency tunability on each slice. This tunability can be software controlled through the provided interfaces.

The RF module handles the interfaces that control the final stage of transmission or first stage of reception of the wireless signals, including antennas, optical telescopes, steerable antennas, external power amplifiers, diplexers, triplexers, RF switches, etc. These external radio equipment components would otherwise be integrated with the RFM except for the physical size and location constraints for transmission and reception. The interfaces are primarily the associated control interfaces for these components. The RFM HID encompasses the control and interface mechanism to the external components. The focus of the RF HID is to provide a standardized interface to the control of each of these devices, to synchronize the operation of the radio with any of these devices.

The other primary capability of the RFM is the conditioning and distribution of the frequency reference as defined by the Frequency Reference Interface. This provides a common reference for the RFM and SPM modules to enable the tracking and navigation functionality typically provided by SDRs. Figure 6, RFM Architecture Details, illustrates the RFM module details.

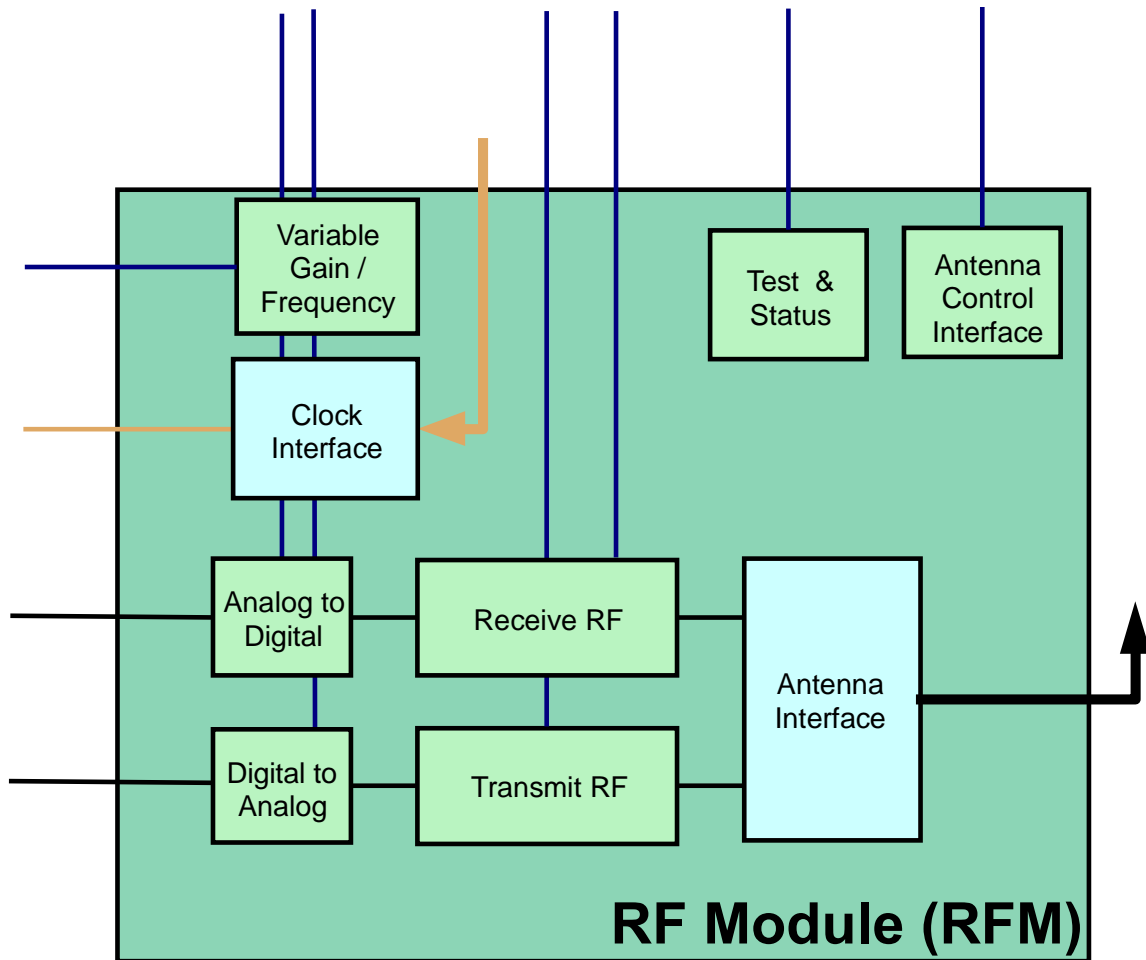


Figure 6—RFM Architecture Details

4.2.3.1 RFM Functions

The RFM transforms the antenna signal to or from a signal usable to the radio. The RFM functions are likely to include the following:

- a. Frequency conversion and gain control.*
- b. Analog filtering.*
- c. Analog-to-digital and digital-to-analog conversion.*
- d. Radiometric tracking.*

4.2.3.2 RFM Components

The RFM can be implemented with a variety of integrated circuits. The control of these circuits can be implemented with a variety of different component technologies including ASICs, discrete electronics, programmable logic devices including FPGAs and DSPs, or even microprocessors. The choice of technologies is left up to the developer of the particular implementation. It is expected that the control of the devices will become more sophisticated over time and that the level of control will increase, resulting in more complex control circuitry and logic devices being used.

4.2.3.3 RFM Interface

- a. *External RF interface(s) to the radio.*
- b. *Provides read and write access to interface registers to monitor and perform control, status, and failure and fault-recovery functions (e.g., via RS-422 or Space Wire).*
 - (1) *Control (power level tunability, frequency tunability, antenna parameter tunability, etc.).*
 - (2) *Status (report status of components and system operation).*
 - (3) *Failure and fault-recovery functions (detect component or system failure and determine appropriate action).*
- c. *Provides diagnostic test registers.*
- d. *Provides I/O for exchanging digitized waveform signal data.*

4.2.3.4 RFM Requirements

(STRS-6) The STRS platform provider shall describe, in the HID document, the behavior and performance of the RF modular component(s).

The behavior and performance of the RF modular components should be sufficiently described such that future waveform developments may take advantage of the RF capability and/or account for its performance. Information in the HID may include such items as center frequency, IF and RF frequency(s), bandwidth(s), IF and RF input/output level(s), dynamic range, sensitivity, overall noise figure, AGC, frequency accuracy and stability, and frequency-tuning resolution.

4.2.4 Security Module

The STRS architecture has been designed to address security concerns as part of the architecture. Although this section is currently not complete, the goal is to address the security services required from an SDR. This approach supports the evolutionary nature of the STRS architecture. It is expected that this section will be expanded as new technologies and operational modes are developed or extended.

The architecture will support selectable data-protection services for those users needing them, including both confidentiality and authentication. Missions may select security options provided by the infrastructure or may develop their own.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

The authentication of commands sent to SDRs is supported, including changing the configuration or uploading new programs for either the infrastructure or new applications. The security section of the architecture will include support for key management, encryption standards, and mitigating threats other than the information and communication security threats currently identified.

4.2.5 Networking Module

The STRS architecture has been structured such that networks can be implemented in an SDR; that is, an SDR can be a node in a network. The SDR may be connected to another node using the appropriate logical and physical interfaces that may be wired and/or wireless. The STRS architecture can accommodate network protocols as services that can be made available to applications and devices. STRS supports the ability to upload new software and dynamic hardware images. Therefore, advancements and replacement of existing protocols can be accomplished without affecting a spacecraft's mission resources.

4.2.6 Optical Module

The STRS architecture supports the use of optical communications in SDRs. The use of optical communications techniques pose challenges in many areas but optical communications also has the potential for great benefit. STRS interfacing to optical communication equipment follows the same techniques shown in integration with high-data-rate hardware. The OM would be controlled through the STRS HAL interface that allows configuration and control of the digital components in the module, which abstracts the optical functionality.

4.3 Hardware Interface Description

The STRS platform provider is to provide an HID, which describes the physical interfaces, functionality, and performance of the entire platform and each platform module. The HID specifies the electrical interfaces, connector requirements, and all physical requirements for the delivered radio. Each module's HID abstracts and describes the module functionality and performance. In this manner, STRS application developers can know the features and limitations of the platform for their applications. The information in the HID provides the knowledge for NASA and others to integrate and test the hardware interfaces. The information in the HID may allow future module replacement or additions without the design of a completely new platform. For example, a Security Module could be added that was not originally planned, or a follow-on mission could use a different frequency band and only require an RFM change.

In addition to the GPM, SPM, and RFM HID descriptions and requirements stated within each module section, the following interface descriptions and requirements are also specified for an STRS platform.

(STRS-4) The STRS platform provider shall describe, in the HID document, the behavior and capability of each major functional device or resource available for use by waveform, services, or other applications (e.g., FPGA, GPP, DSP, or memory), noting any operational limitations.

NASA-STD-4009

(STRS-5) The STRS platform provider shall describe, in the HID document, the reconfigurability behavior and capability of each reconfigurable component.

The description of the behavior and capability of functional devices available to STRS application developers or reconfigurable components may include device type, processing capability, clock speeds, memory size(s), types(s), and speed(s), noting any constraints, as well as any limitation on the number of configurable hardware design reloads, partial reload ability, built-in functionality, and any corresponding restriction on the number of gates.

(STRS-7) The STRS platform provider shall describe, in the HID document, the interfaces that are provided to and from each modular component of the radio platform.

The specific modular components or hardware slices of an STRS radio will vary among different implementations. The STRS platform provider or STRS integrator is expected to describe each modular component and their respective physical and logical interfaces as described in this section. Table 1, STRS Module Interface Characterization, provides typical interface characteristics that should be included when identifying external interfaces or internal interfaces between modules for STRS.

Table 1—STRS Module Interface Characterization

STRS Module Interface Characterization Table	
Parameter	Description and Comments
Name	Interface name (data, control, DC power, RF, security, etc.).
Interface type	Point to point, point-multipoint, multipoint, serial, bus, other.
Implementation level	Component, module, board, chassis, remote node.
Reference documents and standards	Applicable documents for interface standards or description of custom interfaces.
Notes and constraints	Variances from standards, physical and logical functional limitations.
Transfer speed	Clock speed, throughput speed.
Signal definition	Description of functionality and intended use.
Physical Implementation	
Technology	For example, GPP, DSP, FPGA, ASIC, and description.
Connectors	Model number, pin out (incl. unused pins).
Data plane	Width, speed, timing, data encoding, protocols.
Control plane	Control signals, control messages or commanding, interrupts, message protocol.
Functional Implementation	
Models	Data plane model, control plane model, test bench model.
Power	Voltages, currents, noise, conducted immunity, susceptibility.
APIs	Custom or standard, particular to OS environment.
Software	Device drivers, development environment, and tool chain.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

STRS Module Interface Characterization Table	
Parameter	Description and Comments
Logical Implementation	
Addressing	Method, schemes.
Channels	Open, close.
Connection type	Forward, terminate, test.

4.3.1 Control and Data Interface

The control and data communications buses and links between modules within the radio are to be described by the STRS platform provider to the level of detail necessary to facilitate integration of another vendor's module. If modules communicate using the IEEE-1394 interface, for example, this will be specified in the HID with appropriate connector and pinout information. Any nonstandard protocols used should also be specified. In some cases, this may be handled by the software HAL. Module interfaces will be completely described, including any unused pins.

(STRS-8) The STRS platform provider shall describe, in the HID document, the control, telemetry, and data mechanisms of each modular component (i.e., how to program or control each modular component of the platform, and how to use or access each device or software component, noting any proprietary and nonstandard aspects).

Besides the interface descriptions already provided for each modular component, developers should provide specific information necessary for future STRS application developers to know how to interact with the command and control aspects of the platform. The description of the control, telemetry, and data mechanism of each modular component should facilitate the porting of the application software to the platform.

4.3.2 DC Power Interface

The DC power interface description for the radio has two parts: (1) the platform as a supplier to the various modules; and (2) the power consumption of the different modules, if multiple modules are provided.

Table 2, Example—DC Power Interface (Platform Supplied), shows an example listing of a platform DC power interface. There are four distinct sets of power requirements for the platform shown. For each module delivered with the radio, as well as those built by other vendors, the HID is to specify the needed voltages, currents, and connections. Voltages are to be specified with a maximum and minimum tolerance, and associated currents are to be specified with nominal and maximum values. Connectors for DC power are to be specified, including pinouts. If power is routed through a multipurpose connector, such as a backplane connector, then the pins actually used are to be documented. Power is a limited commodity for most missions, and understanding the radio platform power needs is critical.

Table 2—Example—DC Power Interface (Platform Supplied)

Parameter	Values			
Voltage available	–15 V	+2.5 V	+5 V	+15 V
Maximum current/chassis (platform)	2 A	1.7 A	3 A	2 A
Maximum current/slot (module)	1 A	1 A	1 A	1 A
Backplane supply pins	17, 19	59, 61	44, 46, 48	21, 23
Backplane return pins	18, 20	60, 62	43, 45, 47	22, 24
Connector type	-----	-----	-----	-----
Voltage ripple	100 mVpp	1 mVpp	5 mVpp	100 mVpp
Notes	Slot 1 and 2 only	-----	-----	Slot 1 and 2 only

(STRS-9) The STRS platform provider shall describe, in the HID document, the behavior and performance of any power supply or power converter modular component(s).

4.3.3 Thermal Interface and Power Consumption

The power consumption and resulting heat generation of a reprogrammable FPGA will vary according to the amount of logic used and the clock frequency(s). The power consumption may not be constant for each possible waveform that can be loaded on the platform. The STRS platform provider should document the maximum allowable power available and thermal dissipation of the FPGA(s) on the basis of the maximum allowable thermal constraints of FPGA(s) of the platform. For human spaceflight environments, touch temperature requirements may limit dissipation further; therefore, these reductions are to be factored into the given dissipation limits.

(STRS-108) The STRS platform provider shall describe, in the HID document, the thermal and power limits of the hardware at the smallest modular level to which power is controlled.

5. APPLICATIONS

5.1 Application Implementation

As shown in figure 7, Waveform Component Instantiation, an example STRS platform consists of one or more GPMs with GPPs, and optionally one or more SPMs containing DSPs, FPGAs, and ASICs. Application (waveform and service) components loaded and executed on these modules provide the signal-processing algorithms necessary to generate or receive RF signals. To aid portability, the applications are to use the appropriate infrastructure APIs to access platform services. Using “direct to hardware” access instead would increase the effort to port the application to a platform with different hardware. The STRS infrastructure provides the APIs and services necessary to load, verify, execute, change parameters, terminate, or unload an application. The STRS infrastructure utilizes the HAL to abstract communications with the specialized hardware, whereas the HID physically identifies how modules are integrated on a platform.

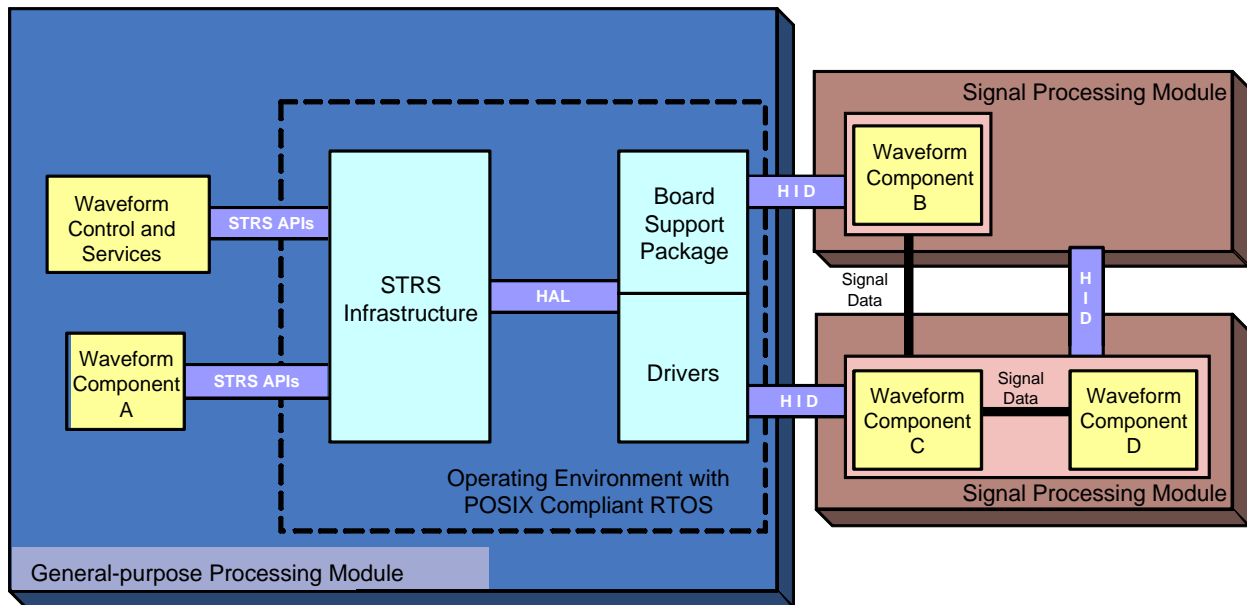


Figure 7—Waveform Component Instantiation

(STRS-10) An STRS application shall use the infrastructure STRS API and POSIX API for access to platform resources.

(STRS-11) The STRS infrastructure shall use the STRS platform HAL APIs to communicate with application components on the platform specialized hardware via the physical interface defined by the STRS platform provider.

5.2 Application Selection

STRS platform providers have the option of providing telemetry values to indicate what types of applications are installed. The method for selecting the application will be a combination of the platform's capabilities as well as the specification defined by the STRS Command and Telemetry interfaces in section 8.

STRS specifies two types of configuration files: a platform-specific component, and an application-specific component. An application-specific configuration file specifies information used to initialize an STRS application. Section 9 contains further discussion of platform and application configuration files.

5.3 Navigation Services

The STRS architecture allows STRS radios to provide radiometric tracking and navigation services that are integrated with communication services. Radiometric tracking is the process of measuring the characteristics of radio signals that have been transmitted (potentially over several legs) in order to extract information relating to the signal's change in frequency and/or time of transit. A radio has the fundamental component needed for tracking—a radio signal. The SDR simplifies the navigation architecture because it minimizes mass, power, and volume

requirements while maximizing flexibility. An SDR provides the flexibility to respond to different mission phase requirements and to dynamic application requirements where signal structures may change. This is the fundamental reason for considering the implementation of an SDR with tracking and navigation functionality.

5.4 Application Repository Submissions

The STRS architecture facilitates the use of reusable and highly reliable applications. Highly reliable and reusable applications require good coding practices, good documentation, and thorough testing. The documentation and application artifacts are to be submitted to the NASA STRS application repository. The use of the artifacts in the NASA STRS application repository will be subject to the appropriate license agreements. Therefore, the agreements defining the release, distribution, and ownership of the artifacts are to be submitted to the repository including license agreements, type of release, and any restrictions. Types of releases are discussed in NPR 2210.1, Release of NASA Software. NASA will provide the STRS application developer information on the requests and distribution of items and lessons learned using the application. If the STRS application developer receives independent requests for the application, this request should be forwarded to the NASA STRS application repository manager to assure process consistency.

The goal of the NASA STRS application repository is to reduce future application development time and porting time since STRS application developers will have access to validated code. The STRS application repository is an archive of the developed configurable hardware design and software for the various applications. The repository allows STRS application developers access to existing STRS application artifacts that have been populated by NASA and STRS application developers. The documentation of STRS application behavior should include the STRS application developer's implementation of the STRS Application-provided Application Control API methods as described in section 7.3.1.

(STRS-12) The following application development artifacts shall be submitted to the NASA STRS application repository.

- (1) High-level system or component software model.
- (2) Documentation of application configurable hardware design external interfaces (e.g., signal names, descriptions, polarity, format, data type, and timing constraints).
- (3) Documentation of STRS application behavior.
- (4) Application function sources (e.g., C, C++, header files, very high speed integrated circuit HDL (VHDL), and Verilog).
- (5) Application libraries, if applicable (e.g., electronic design interchange format (EDIF), dynamic link library (DLL)).
- (6) Documentation of application development environment and tool suite.
 - A. Include application name, purpose, developer, version, and configuration specifics.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

- B. Include the hardware on which the application is executed, its OS, OS developer, OS version, and OS configuration specifics.
 - C. Include the infrastructure description, developer, version, and unique implementation items used for application development.
- (7) Test plans, procedures, and results documentation.
 - (8) Identification of software development standards used.
 - (9) Version of this Standard used.
 - (10) Information, along with supporting documentation, required to make the appropriate decisions regarding ownership, distribution rights, and release (technology transfer) of the application and associated artifacts.
 - (11) Version Description Document or equivalent with version numbers defined down to the lowest level components.
 - (12) Documentation of the platform component hardware used by the application, its function and the interconnections. If the component executes an operating system, document the OS, OS developer, OS version, and OS configuration.

6. CONFIGURABLE HARDWARE DESIGN ARCHITECTURE

Configurable hardware design is embedded in a hardware device, such as an FPGA. Configurable hardware design is distinguished from software residing in a GPP, which is generally easier to change. This section addresses the use of configurable hardware design from design and development through testing and verification and operations. It addresses aspects of model-based design techniques and design for space environment applications.

Proper testing of configurable hardware design is critical in the development of reliable and reusable code. Development tools that enable early development and testing should be used so that problems can be identified and resolved early in the SDR life cycle. Many real-world signal degradations and SEUs can be simulated to identify potential issues with the waveform and waveform functions early in development, even before hardware is available. Applications implemented in configurable hardware should be modular with clear interfaces to enable individual application component simulations and incremental testing.

The configurable hardware design architecture supports the modeling of STRS applications implemented in configurable hardware at the system, subsystem, and function levels. Model-based design techniques aid in the development of modular application functions. Application development models done in a platform (or target) independent manner aid in application testing, reuse, and portability. A platform-independent model (PIM) design can be used to target different platforms. PIM design flows might include high level models combined with manual code writing. On resource-constrained platforms, optimized code would be written. On non-resource-constrained platforms, PIMs may be used to auto generate code. These design flows can be employed to significantly reduce the porting effort.

Application portability should be considered in all facets of the design process from concept to implementation to testing. The coding technique of the application is also essential to reduce the application porting effort. Having defined syntax standards for HDLs (e.g., Verilog or VHDL) makes them appear to be easily portable across devices and software synthesizers, but this is an incorrect assumption. There are many things that can make hardware description languages hard to port. For example, the use of device-specific fixed hardware logic on the FPGA will decrease the portability. The use of specialized hardware may be required to meet the timing constraints of the application; however, the STRS application developer should document any application function that uses the specialized hardware so that the effort to port the application function(s) can be determined. Non-boolean-type logic such as clock generation can also reduce portability. One method to decrease the porting effort would be to create a module that does the clock generation from which the rest of the application functions receive the necessary clock(s).

Development of configurable hardware design for STRS radios should include provisions for mitigating space environmental effects such as SEUs. Near-term application of static random access memory (SRAM)-based FPGAs may require triple-mode redundancy (TMR), configuration memory scrubbing, and other mitigation techniques, depending on the intended mission environment and desired reliability. Commercial design tools are becoming available to aid in this process and some newer FPGAs have versions available with embedded TMR.

A key feature of SDRs is that they can be reconfigured after deployment. The ability to load new applications and services will benefit missions in several ways, including using one SDR (instead of several separate radios) to handle different applications for various phases of a mission, some planned and some unplanned. An STRS platform should receive STRS application software and configurable hardware design updates after deployment.

6.1 Specialized Hardware Interfaces

Standardizing and documenting the interface from the waveform applications on the GPP to the portion of the waveform in the specialized processing hardware, such as FPGAs, is intended to provide commonality among different STRS platforms and to aid portability of application functional components implemented in configurable hardware design. Figure 8, High-Level Software and Configurable Hardware Design Waveform Application Interfaces, depicts the high-level interface relationship between GPM, SPM, and RFM modules in an STRS radio.

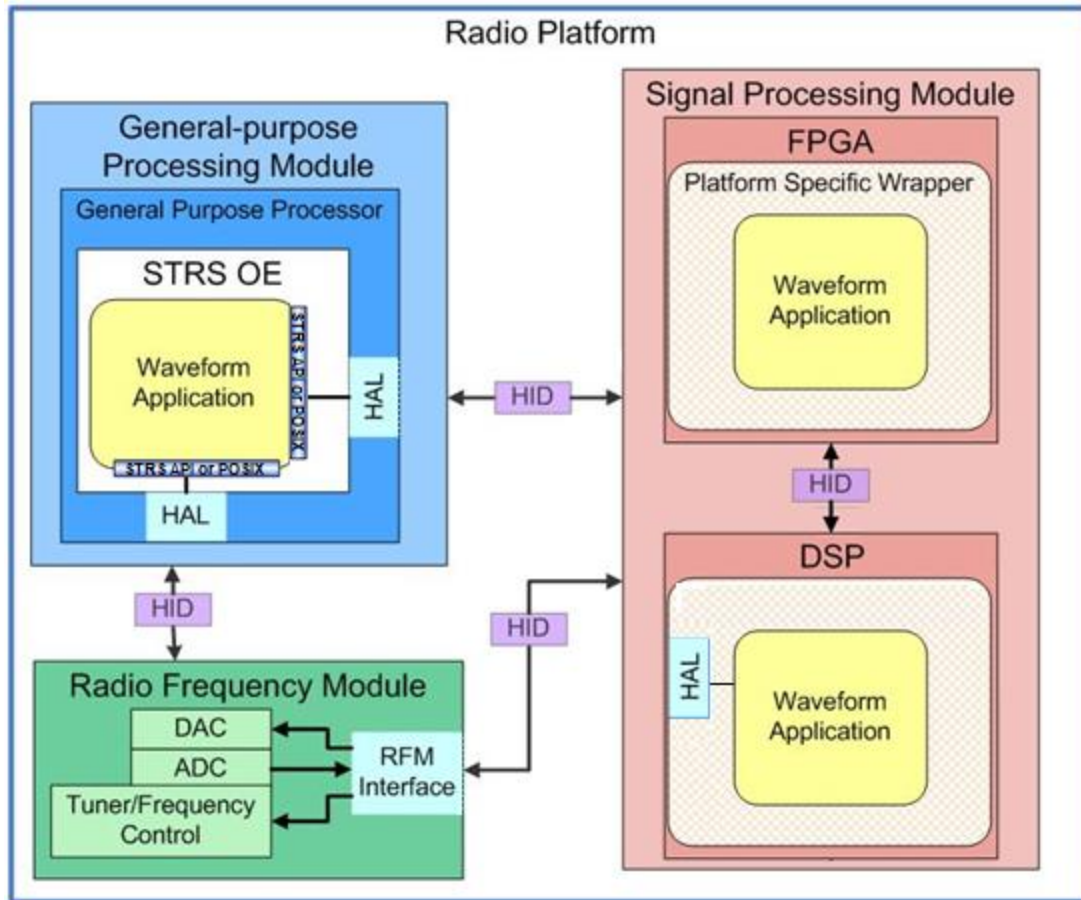


Figure 8—Notional High-Level Software and Configurable Hardware Design Waveform Application Interfaces

The STRS architecture provides a common mechanism for the software to instantiate, configure, and execute the software and configurable hardware design applications on various platforms using different hardware devices. Reconfiguration may include changing the parameters of installed applications and uploading new applications after deployment.

The application accepts configuration and control commands from the GPM and uses STRS APIs or POSIX APIs that interface to the device drivers associated with the SPM and RFM modules. The device drivers communicate via the HAL on the GPM that abstracts the physical interface specification described in the HID in transferring command and data information between the modules.

For FPGAs, the interface to the application is through a platform-specific wrapper. The platform-specific wrapper accepts command and data information from the GPM and provides them to the application. The platform-specific wrapper also abstracts details of the platform from the STRS application developer, such as pinout information. The platform-specific wrapper should also provide clock generation, signal registering, and synchronization functions, and any other non-waveform-specific functions that the platform requires.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

Documentation of the platform-specific wrapper is necessary so that STRS application developers can interface applications to the platform. This documentation should include detailed timing constraints, such as signal hold times, minimum pulse widths, and duty cycles. The signal timing constraints refer to the protocol of a particular interface describing events happening on a particular clock cycle. For clock generation, one should document what clock domains are in the design, how each clock is generated, and the resources that are required. Signal synchronization describes any additional logic needed when clock domains are changed across the interface. The signal registering methods refer to any configurable hardware design interfaces between modules and if the input and output were registered, latched, or neither.

(STRS-13) If the STRS application has a component resident outside the GPM (e.g., in configurable hardware design), then the component shall be controllable from the STRS OE.

(STRS-14) The STRS SPM developer shall provide a platform-specific wrapper for each user-programmable FPGA, which performs the following functions:

- (1) Provides an interface for command and data from the GPM to the waveform application
- (2) Provides the platform-specific pinout for the STRS application developer. This may be a complete abstraction of the actual FPGA pinouts with only waveform application signal names provided.

(STRS-15) The STRS SPM developer shall provide documentation on the configurable hardware design interfaces of the platform-specific wrapper for each user-programmable FPGA, which describes the following:

- (1) Signal names and descriptions.
- (2) Signal polarity, format, and data type.
- (3) Signal direction.
- (4) Signal-timing constraints.
- (5) Clock generation and synchronization methods.
- (6) Signal-registering methods.
- (7) Identification of development tool set used.
- (8) Any included noninterface functionality.

7. SOFTWARE ARCHITECTURE

7.1 Software Layer Interfaces

The STRS architecture is predicated on the need to provide a consistent and extensible development environment on which to construct NASA space applications. The breadth of this goal requires that the specification be based on the following: (1) Core interfaces that allow flexibility in the development of application software; and (2) HIDs that enable technology infusion.

The software architecture model shows the relationship between the software layers expected in an STRS-compliant radio. The model illustrates the different software elements used in the software execution and defines the software interface layers between applications and the OE and the interface between the OE and the hardware platform.

Figure 9, STRS Software Execution Model, represents the software architecture execution model. The software model achieves the following objectives:

- a. Abstracts the application from the underlying OE software to promote portability of the application.*
- b. Within the abstraction layer, minimizes custom routines by using commercial software standard interfaces such as POSIX.*
- c. Depicts the STRS software components as layers to specify their relationship to each other and their separation from each other which enables developers to implement the layers differently according to their needs while still complying with the architecture.*
- d. Introduces a lower-level abstraction layer between the OE and the platform hardware.*

Note that although software abstraction for general processors is typically accomplished with board support packages and device drivers, the abstraction of hardware languages or configurable hardware design is less defined. The model represents the software and configurable hardware design abstraction in this layer.

- e. Indicates the relationship between the OE software and the different hardware processing elements (e.g., processor and specialized hardware).*

The OE adheres to the interface descriptions provided in figure 9. This Standard, provides two primary interface definitions, as follows: (1) The STRS API; and (2) The STRS HAL specification, each with a control and data plane specification for interchanging configuration and run-time data. The STRS APIs provide the interfaces that allow applications to be instantiated and use platform services. These APIs also enable communication between application components. The HAL specification describes the physical and logical interfaces for intermodule and intramodule integration.

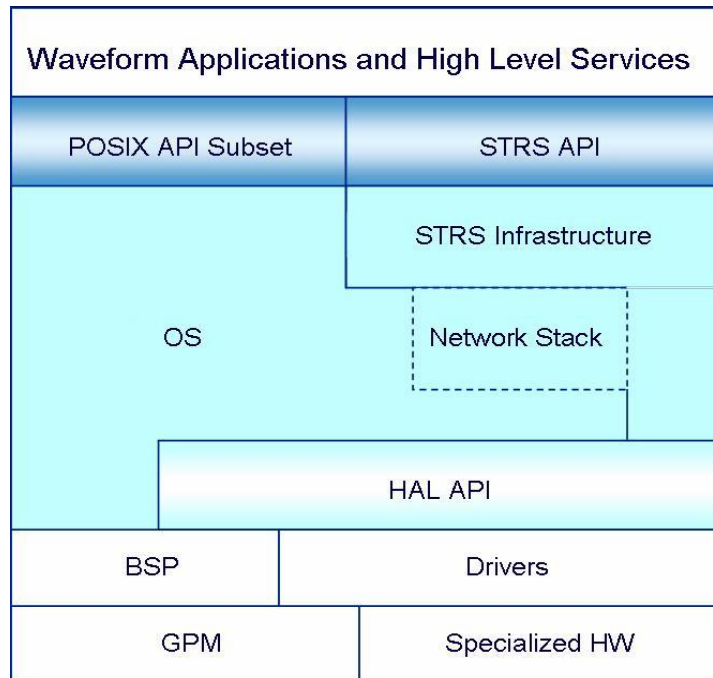


Figure 9—STRS Software Execution Model

The STRS software architecture presents a consistent set of APIs to allow waveform applications, services, and communication equipment to interoperate in meeting an application specification. Figure 10, STRS Layered Structure in UML, represents a view of the platform OE that depicts the boundaries between the STRS infrastructure provided by the STRS platform provider and the components that can be developed by third-party vendors (e.g., waveform applications and services).

A key enabler of application portability is the removal of application dependencies on the infrastructure that take advantage of explicit knowledge of the infrastructure implementation. When waveforms and services conform to the API specification, they are easier to port to other STRS platform implementations.

Figure 10 extends the view of the software architecture from the diagram introduced in figure 9 to include additional detail of the infrastructure, POSIX-conformant OS, and hardware platform. The STRS Software Execution Model (figure 9) was transformed using the Unified Modeling Language (UML). The UML supports the description of the software systems using an object-oriented style. This approach clarifies the interfaces between components, adding additional detail. Table 3, STRS Architecture Subsystem Key, provides a key that describes the interaction between elements of the architecture.

Figure 11, STRS Operating Environment, describes the elements of the detailed OE depicted in figure 9. In the case that the OS does not support the POSIX subset, the missing functionality is to be implemented in the STRS infrastructure. Figure 11 also illustrates the inclusion of a POSIX abstraction layer in the infrastructure. As a note, this abstraction is not only for a non-POSIX OS, but the POSIX abstraction layer would implement any POSIX functions required but not implemented by the OS.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

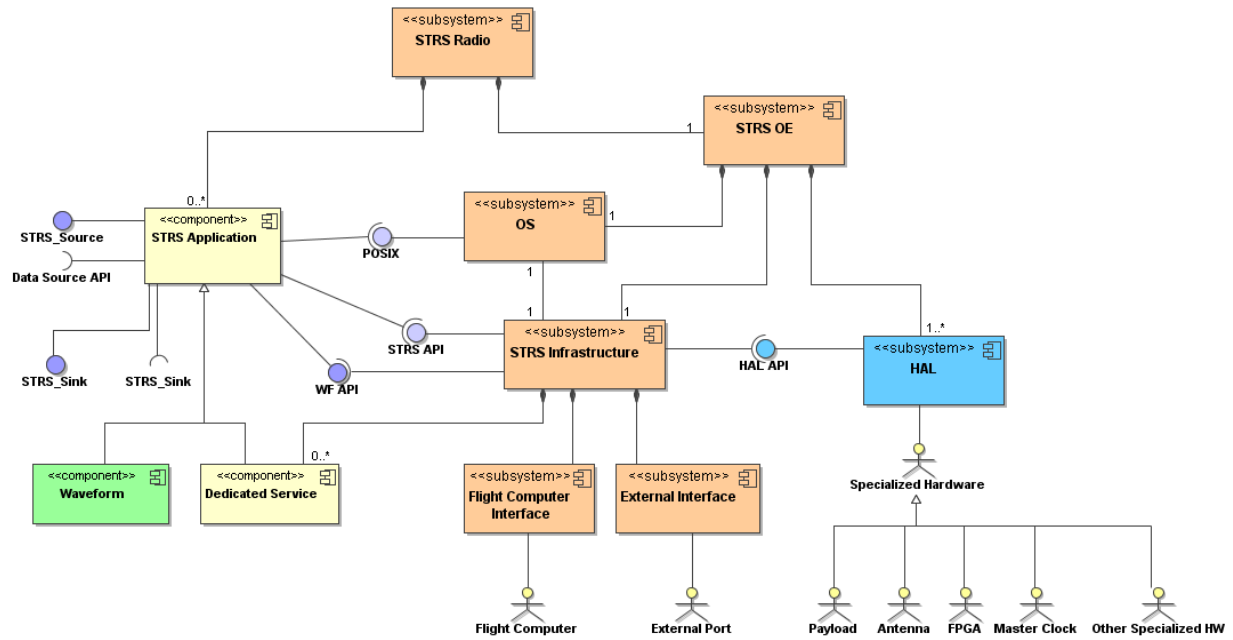

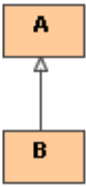
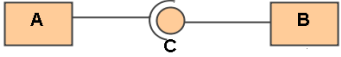
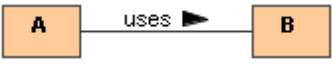



Figure 10—STRS Layered Structure in UML

Table 3—STRS Architecture Subsystem Key

Diagram Element	Name	Explanation
	Composition	A contains X items of type B. B is a part of the aggregate A. B does not exist independently from A. X may be a number or a range from <i>m</i> to <i>n</i> depicted by “ <i>m...n</i> ” where <i>n</i> may be an asterisk to indicate no upper limit.
	Generalization or Inheritance	B is derived from A. B is a kind of A. B inherits all the properties of A. A is a more general case of B. Since B is more specialized, it frequently contains some additional attributes and/or more functionality than A.
	Interface	C is an interface provided by B; that is, C contains the means to invoke behavior that resides in B. A uses interface C to access B.
	Association	A is associated with B. The optional description “uses” indicates that A is associated with B such that A “uses” B.
	Association	D acts upon A, and A responds to D, or possibly vice versa. D is normally an actor outside the system.

In figure 11 the arrows identify interface dependencies and isolations. The waveform applications will not directly call the driver API but use the provided STRS API, thus providing the “abstraction layer” that helps isolate the application from the platform.

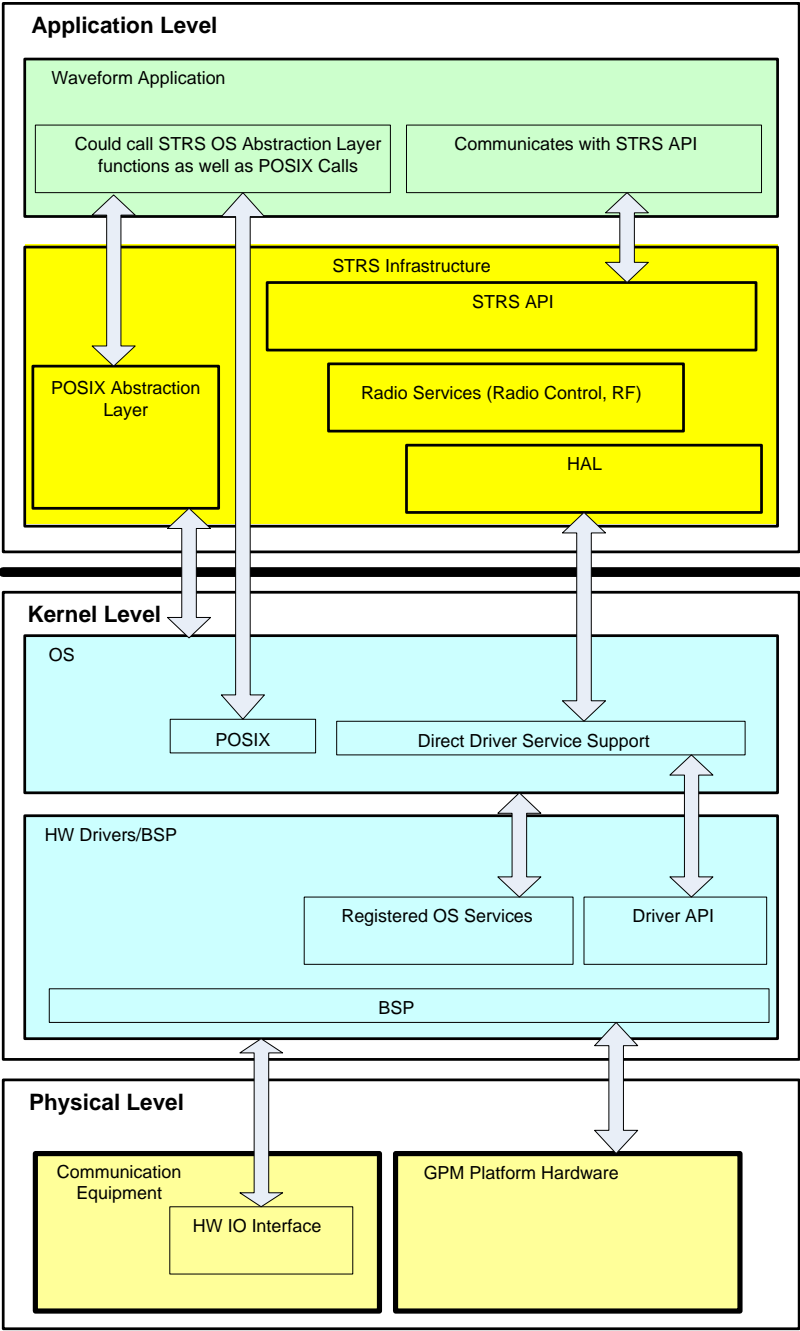


Figure 11—STRS Operating Environment

In table 4, STRS Software Component Descriptions, the different layers of the STRS software model are described.

Table 4—STRS Software Component Descriptions

Layer	Description
Waveform application and services	Waveform application and services provide the radio GPP functionality using the STRS infrastructure.
STRS infrastructure	The STRS infrastructure implements the behavior and functionality identified by the STRS API as well as other required radio functionality.
STRS API	The STRS API provides consistent interfaces for the STRS infrastructure to control applications and services, and for the applications and services to access STRS infrastructure services.
APP API	The APP API is the interface implemented by waveforms and services whose functions are used by the STRS infrastructure.
POSIX Abstraction Layer	This optional interface (see figure 12, POSIX-Compliant Versus POSIX-Conformant OS) provides POSIX OS services to the waveform application and services on platforms with an OS that does not provide POSIX interfaces.
Radio control services	These services are responsible for handling the radio commands and telemetry for the STRS. Applications use the STRS interface to communicate telemetry and receive commands from flight computer.
HAL	The HAL provides the device control interfaces that are responsible for all access to the hardware devices in the STRS radio. The HAL API is the interface to the software drivers and BSP that communicates with the hardware.
POSIX API	The STRS defines a minimum POSIX AEP for the allowed OS services. The POSIX AEP can be implemented by either a POSIX-conformant OS or by a POSIX Abstraction Layer in conjunction with a nonconformant OS.
OS	This is the operating system that supports the POSIX API and other OS services. The POSIX Abstraction Layer will provide applications with a consistent AEP interface that is mapped into the chosen OS functions.
POSIX OS	This is the STRS POSIX AEP-conformant portion of the OS.
Direct service support	This layer identifies the ability for the STRS infrastructure to have a direct interface to the hardware drivers on the platform.
HW drivers/BSP	The hardware drivers provide the platform independence to the software and infrastructure by abstracting the physical hardware interfaces into a consistent device control API.
Registered OS services	These are services that are integrated with the chosen OS to provide services such as MAC-layer interface to physical Ethernet hardware.
Driver API	OS-supplied APIs are abstracted from applications via the device control API.

NASA-STD-4009

Layer	Description
BSP	The BSP is the software that implements the device drivers and parts of the kernel for a specific piece of hardware. It provides the hardware abstraction of the GPM module for the POSIX-compliant OS. A BSP contains source files, binary files, or both. A BSP contains an original equipment manufacturer (OEM) adaptation layer (OAL), which includes a boot loader for initializing the hardware and loading the OS image. Essentially, the OAL is all of the software that is hardware specific. The OAL is actually compiled and linked into the embedded OS.
HW I/O interfaces	Device drivers have been created for these physical interfaces.
GPM	This is the general-purpose processing module on which the STRS infrastructure executes.
Specialized hardware	This is the physical layer of the hardware modules existing on the STRS platform.

Figure 12 illustrates the difference between a POSIX-conformant OS and a nonconformant OS. On the left side, the POSIX AEP is provided entirely by the OS. The POSIX APIs are included in those for the OS. On the right side, the OS is not POSIX AEP conformant but is partially compliant. The POSIX AEP is shown in two parts. One part shows the POSIX APIs that are included in the OS. The other part shows the part of the POSIX AEP that is not provided by the OS but is to be provided as the POSIX abstraction layer. The STRS OE includes a POSIX PSE51-conformant OS or POSIX abstraction layer for missing APIs.

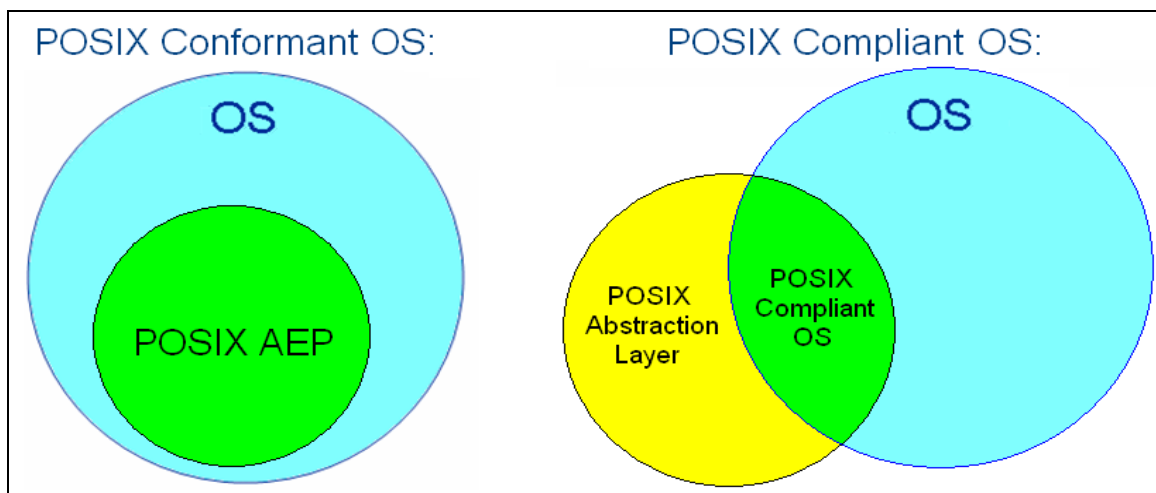


Figure 12—POSIX-Compliant Versus POSIX-Conformant OS

7.2 Infrastructure

The STRS infrastructure is part of the OE and provides the functionality for the interfaces defined by the STRS API specification. The infrastructure exposes a standard set of method names to the applications to facilitate portability. Although the STRS infrastructure may use any combination of POSIX, OS, BSP functions, or other infrastructure methods to implement a radio function, which may vary on different platforms, the STRS API will be the same to allow portability. The STRS API is the well-defined set of interfaces used by STRS applications to access specific radio functions or used by the infrastructure to control the applications.

The infrastructure is composed of multiple subsystems that interoperate to provide the functionality to operate the radio. The components shown in figure 13, STRS Infrastructure, represent the high-level subsystems and services needed to control STRS applications within the radio platform. These services are provided by the platform infrastructure and support applications as they execute within the radio platform. The infrastructure functions will include fault management techniques, which are necessary to increase radio robustness and support mission-dependent requirements. In order to support one of the primary objectives of the STRS (upgradeability), an STRS radio should be able to receive updated versions of the OE to support applications developed for newer versions of this Standard, after deployment.

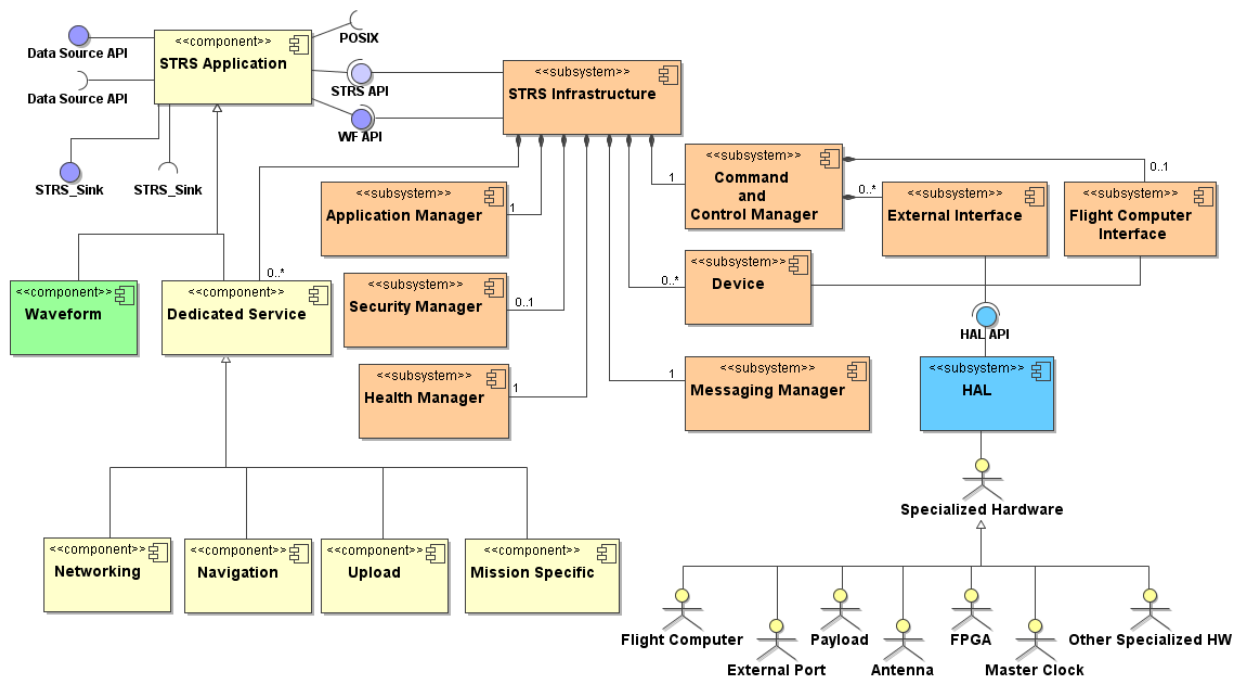


Figure 13—STRS Infrastructure

7.3 STRS APIs

The STRS APIs provide an open software specification so that the application engineers can develop STRS applications. The goal is to have a standard API available to cover all application program requirements so that the application programs can be reused on other hardware systems with minimal porting effort and cost for the application implemented in software and/or configurable hardware design with increased reliability. Size, weight, and power constraints may limit the functionality of the radio by imposing a tradeoff among the following: (1) The size of the API implementation; (2) The size of other internal operations; and (3) The size of the waveforms and services. The size of the selected GPP should be sufficient to contain the OS, the STRS infrastructure, and the appropriate portion of the waveforms and services to implement the required mission functionality, along with sufficient margin to support software upgrades. The STRS APIs are defined to support internal radio commands. The external interface commands, described in section 9, often use the internal commands defined by the STRS APIs to accomplish normal radio operations.

The API layer specification decouples the intellectual property rights of platform, application, and module developers. The API layer allows development and interoperability of different radio aspects while protecting the investment of the developers. The definitions of the APIs are based on a set of sequence diagrams derived from the use cases identified in Appendix B of the NASA/TP-2008-214813, STRS Software Architecture Concepts and Analysis, document.

The APIs are defined in the following sections. The APIs are grouped by type to simplify the description of the APIs while providing the detail for each requirement in tabular form. The table contains the name, description, calling sequence, return type, any preconditions, any post conditions, and examples. The examples shown in the table for each requirement are written from the point of view of the STRS application developer. The calling sequences for the infrastructure-provided APIs are callable from C language implementations of the STRS applications. If coding is done in C++, the infrastructure-provided API methods do not belong to any class and should be defined using extern "C."

A "handle ID" is an identifier used to control access to applications, devices, files, messaging queues, and other similar resources. The same handle name refers to the same application, device, file, queue, timer, or service across all applications. For information about errors, see section 7.3.11.

(STRS-105) The STRS infrastructure APIs shall have an ISO/IEC C language compatible interface.

7.3.1 STRS Application-Provided Application Control API

A key aspect of a software-architecture is the definition of the APIs that are used to facilitate software configuration and control of the target platform. The philosophy on which the STRS architecture is based avoids the conflict between open architecture and proprietary implementations by specifying a minimum set of APIs that are used to execute waveform applications and to deliver data and control messages to installed hardware components.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

The following APIs exhibit similar functionality to a resource interface in the Object Management Group (OMG)/software radio (SWRADIO) or Software Communications Architecture (SCA). The APIs could be implemented using the same Platform-Independent Model (PIM) as the OMG/SWRADIO or SCA and a different platform-specific model (PSM) from the OMG/SWRADIO or SCA. The APIs are further grouped similar to the OMG/SWRADIO as shown in figure 14, STRS Application and Device Structure.

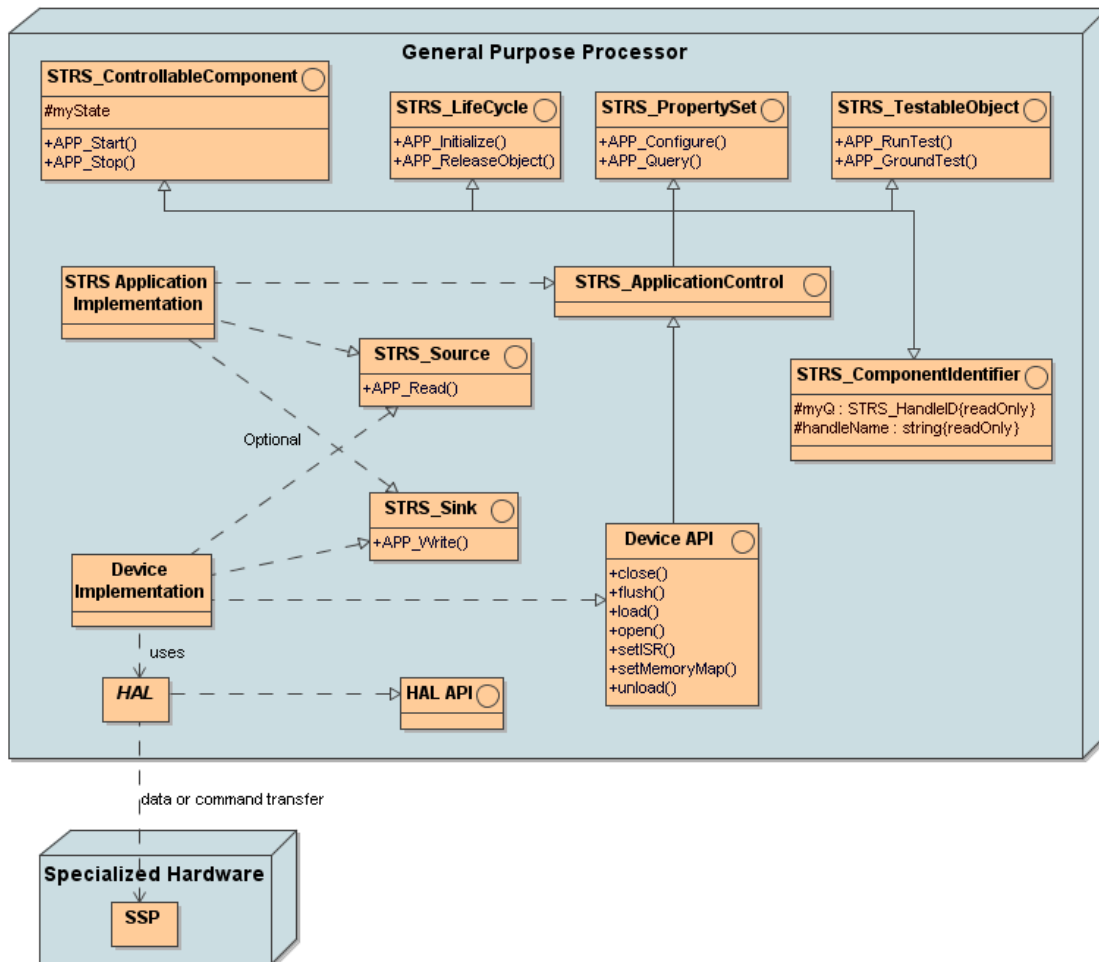


Figure 14—STRS Application and Device Structure

As shown in figure 14, an STRS application implementation (e.g., waveform) is derived from the STRS_ApplicationControl API, the STRS_Source API when implementing APP_Read, and the STRS_Sink API when implementing APP_Write. The interfaces are implemented in groups so that STRS_ApplicationControl is derived from the STRS_LifeCycle, STRS_PropertySet, STRS_TestableObject, STRS_ControllableComponent, and STRS_ComponentIdentifier interfaces.

NASA-STD-4009

STRS requires a C and C++ standard based on ISO/IEC 9899 and ISO/IEC 14882, respectively. In the USA, this is INCITS/ISO/IEC 9899:year and INCITS/ISO/IEC 14882:year, respectively, where the year will change periodically. The year is not included in the requirement so that obsolete compilers are not mandated. In the USA, the InterNational Committee for Information Technology Standards (INCITS) coordinates technical standards activity between ANSI in the USA and joint ISO/IEC committees worldwide. INCITS is not included in the requirement, so that the country of implementation may use its compilers.

(STRS-16) The STRS Application-provided Application Control API shall be implemented using ISO/IEC C or C++.

(STRS-17) The STRS infrastructure shall use the STRS Application-provided Application Control API to control STRS applications.

An OE may support applications written in either C, C++, or both. An application written for an OE that supports only C++ will require extra effort to port it to an OE that supports only C and vice versa.

(STRS-18) The STRS OE shall support ISO/IEC C or C++, or both, language interfaces for the STRS Application-provided Application Control API at compile-time.

(STRS-19) The STRS OE shall support ISO/IEC C or C++, or both, language interfaces for the STRS Application-provided Application Control API at run-time.

The same include files are used for either C or C++ to access the appropriate prototypes.

(STRS-20) Each STRS application shall contain
#include "STRS_ApplicationControl.h"

(STRS-21) The STRS platform provider shall provide an "STRS_ApplicationControl.h" that contains the method prototypes for each STRS application and, for C++, the class definition for the base class STRS_ApplicationControl.

(STRS-22) If the STRS Application-provided Application Control API is implemented in C++, the STRS application class shall be derived from the STRS_ApplicationControl base class. *For example, the MyWaveform.h file should contain a class definition of the form class MyWaveform: public STRS_ApplicationControl {...};*

A sink is used for a push model of passing data, that is, to write data to the waveform, device, file, or queue.

(STRS-23) If the STRS application provides the APP_Write method, the STRS application shall contain
#include "STRS_Sink.h"

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

(STRS-24) The STRS platform provider shall provide an “STRS_Sink.h” that contains the method prototypes for APP_Write and, for C++, the class definition for the base class STRS_Sink.

(STRS-25) If the STRS Application-provided Application Control API is implemented in C++ and the STRS application provides the APP_Write method, the STRS application class shall be derived from the STRS_Sink base class.

For example, the MyWaveform.h file should contain a class definition of the form

```
class MyWaveform: public STRS_ApplicationControl,  
                  public STRS_Sink  
{...};
```

A source is used for a pull model of passing data: to read data from the waveform, device, file, or queue.

(STRS-26) If the STRS application provides the APP_Read method, the STRS application shall contain

```
#include "STRS_Source.h"
```

(STRS-27) The STRS platform provider shall provide an “STRS_Source.h” that contains the method prototypes for APP_Read and, for C++, the class definition for the base class STRS_Source.

(STRS-28) If the STRS Application-provided Application Control API is implemented in C++ and the STRS application provides the APP_Read method, the STRS application class shall be derived from the STRS_Source base class.

For example, the MyWaveform.h file should contain a class definition of the form

```
class MyWaveform: public STRS_ApplicationControl,  
                  public STRS_Source  
{...};
```

If both APP_Read and APP_Write are provided in the same waveform, the C++ class will be derived from all three base classes named in requirements (STRS-22, STRS-25, and STRS-28).

For example, the MyWaveform.h file should contain a class definition of the form

```
class MyWaveform: public STRS_ApplicationControl,  
                  public STRS_Sink,  
                  public STRS_Source  
{...};
```

The following state diagram, figure 15, STRS Application State Diagram, shows that an STRS application can have various states during execution. The files for the STRS application are to be accessible before execution can begin.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

- *STRS_InstantiateApp* causes the deployed configuration file to be parsed and *APP_Instance* or the constructor to be called such that the STRS application starts in the *STRS_APP_INSTANTIATED* state, but it may be transitioned to another state if specified in the STRS application configuration file.
- *STRS_Initialize* calls *APP_Initialize* on the appropriate STRS application.
- *APP_Initialize* transitions the STRS application to the *STRS_APP_STOPPED* state upon successful completion.
- *STRS_Start* calls *APP_Start* on the appropriate STRS application.
- *APP_Start* transitions the STRS application from the *STRS_APP_STOPPED* state to the *STRS_APP_RUNNING* state upon successful completion.
- *STRS_Stop* calls *APP_Stop* on the appropriate STRS application.
- *APP_Stop* transitions the STRS application from the *STRS_APP_RUNNING* state to the *STRS_APP_STOPPED* state upon successful completion.
- *STRS_ReleaseObject* calls *APP_ReleaseObject* on the appropriate STRS application.
- The *FAULT* state may be set by the STRS application or detected by the fault monitoring and recovery functions, but any recovery is managed by the STRS infrastructure or by an external system.

*The STRS application states shown in figure 15 are the only ones returned when requested by a call to *APP_RunTest* with a test ID of *STRS_TEST_STATUS*. The STRS application developer may define and use any additional internal substates that the STRS application developer sees fit; however, these substates are not recognized by the infrastructure. The infrastructure may use any additional states that are deemed necessary.*

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

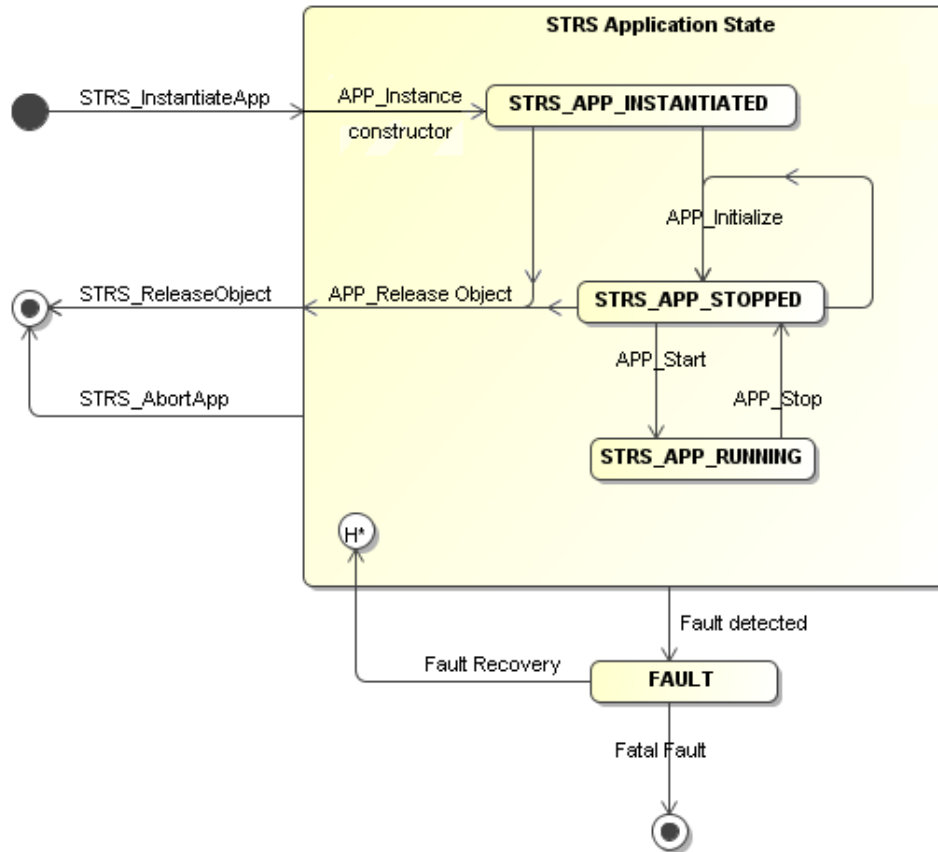


Figure 15—STRS Application State Diagram

NASA-STD-4009

The following are the STRS Application-provided Application Control APIs:

Table 5—APP_Configure()

APP_Configure()	
Description	Set values for one or more properties in the application. It is the responsibility of the application (or device) to determine which properties can be changed in which states. The caller manages the propList, preallocating and filling the names and values before calling APP_Configure. The API is defined in STRS_PropertySet. The method is similar to configure() in PropertySet interface in SCA or OMG/SWRADIO.
Parameters	<ul style="list-style-type: none"> propList - (in STRS_Properties *) list of name and value pairs
Return	status (STRS_Result)
Precondition	Storage for the propList with space for sufficient name and value pairs; sufficient space for each name and value is allocated before calling APP_Configure.
Postcondition	The appropriately named values are configured. The state is unchanged unless specifically required by the mission.
See Also	STRS_Configure
Example	<pre> STRS_Result APP_Configure(STRS_Properties * propList) { STRS_Result rtn = STRS_OK; int ip; for (ip=0; ip<propList->nProps, ip++) { if (strcmp("A", propList->vProps[ip].name)==0){ strncpy(a, propList->vProps[ip].value, maxLa); } else if (strcmp("B", propList->vProps[ip].name)==0){ if (myState == STRS_APP_RUNNING) { rtn = STRS_WARNING; } else { strncpy(b, propList->vProps[ip].value, maxLb); } } else { rtn = STRS_WARNING; } } return rtn; } </pre>

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Table 6—APP_GroundTest()

APP_GroundTest()	
Description	Perform unit and system testing, which is usually done before deployment. The testing may include calibration. The tests aid in isolating faults within the application. This method provides more exhaustive testing that is required before entrusting life and property to an SDR. Application may be in any state, but certain tests may be restricted to specific states. Property values may be specified or retrieved. The propList may be NULL if it is not used. The caller manages the propList, preallocating the structure. The caller fills in the appropriate list of names and any input values and sets nProps to the number of names in the list (nProps > 0). The application fills in any output values for those Properties whose names are specified in the propList. The API is defined in STRS_TestableObject. The method is similar to APP_RunTest except that it contains more extensive testing that will be eliminated for actual flight. This method may be invalid upon deployment.
Parameters	<ul style="list-style-type: none"> testID - (in STRS_TestID) number of the test to be performed propList - (inout STRS_Properties*) list of the name and value pairs used to configure the test, and/or return results.
Return	status or state (STRS_Result)
Precondition	The propList is to have space allotted for the maximum number of properties whose values are to be used or returned.
Postcondition	The test is performed. The state is unchanged unless specifically required by mission.
See Also	STRS_GroundTest
Example	<pre> STRS_Result APP_GroundTest(STRS_TestID testID, STRS_Properties *propList) { if (testID == 0) { ... return STRS_OK; } else { STRS_Buffer_Size nb = strlen("Invalid APP_GroundTest argument."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "Invalid APP_GroundTest argument.", nb); return STRS_ERROR; } } </pre>

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Table 7—APP_Initialize()

APP_Initialize()	
Description	Initialize the application. The API is defined in STRS_LifeCycle. The method is similar to initialize() in LifeCycle interface in SCA or OMG/SWRADIO. The purpose is to set or reset the application to a known initial state. If no fault is detected, this method changes the state to STRS_APP_STOPPED state.
Parameters	None
Return	status (STRS_Result)
Precondition	Application is in STRS_APP_INSTANTIATED or STRS_APP_STOPPED state.
Postcondition	Application is in the STRS_APP_STOPPED state.
See Also	STRS_Initialize
Example	<pre> STRS_Result APP_Initialize() { if (myState == STRS_APP_RUNNING) { STRS_Buffer_Size nb = strlen("Can't Init when STRS_APP_RUNNING."); STRS_Log(fromWF, STRS_WARNING_QUEUE, "Can't Init when STRS_APP_RUNNING.", nb); return STRS_WARNING; } else { ... myState = STRS_APP_STOPPED; } return STRS_OK; } </pre>

NASA-STD-4009

Table 8—APP_Instance()

APP_Instance()	
Description	Store the two parameters passed in the calling sequence, handle ID identifier in myQ, and handle name in handleName, respectively, so that they are available to the other methods in the application. In C++, APP_Instance is a static method used to call the class constructor for C++. If no fault is detected, this method returns an instance pointer and changes the state to STRS_APP_INSTANTIATED state.
Parameters	<ul style="list-style-type: none"> • id - (in STRS_HandleID) handle ID of this STRS application. • name – (in char*) handle name of this STRS application.
Return	Pointer to instance of class, in C++. Non-null, in C.
Precondition	Any.
Postcondition	The application is in the STRS_APP_INSTANTIATED state.
See Also	N/A.
Example for C++	<pre> ThisSTRSApplication *ThisSTRSApplication::APP_Instance(STRS_HandleID handleID, char *name) { return new ThisSTRSApplication(handleID,name); } </pre>
Example for C	<pre> char handleName[nMax]; ThisSTRSApplication *APP_Instance(STRS_HandleID handleID, char *name) { myQ = handleID; strncpy(handleName, name, nMax); myState = STRS_APP_INSTANTIATED; return name; } </pre>

NASA-STD-4009

Table 9—APP_Query()

APP_Query()	
Description	Obtain values for one or more properties in the application. The caller manages the propList, preallocating the structure. The propList may not be NULL. If the caller fills in the appropriate list of names and sets nProps to the number of names in the list (nProps > 0), only those values will be returned whose names are specified in the propList. If the caller specifies no names in propList (nProps = 0), both names and values are filled in up to the maximum number allotted (mProps). The API is defined in STRS_PropertySet. The method is similar to query() in the PropertySet interface in SCA or OMG/SWRADIO.
Parameters	<ul style="list-style-type: none"> propList - (inout STRS_Properties *) - list of name and value pairs
Return	status (STRS_Result)
Precondition	The propList is to have space allotted for the maximum number of properties whose values are to be returned.
Postcondition	propList is populated with values if names are already in the list (if nProps > 0), or else populated with all available names and values up to the maximum (mProps).
See Also	STRS_Query
Example	<pre> STRS_Result APP_Query(Properties *propList) { int ip; if (propList == NULL) { STRS_Buffer_Size nb = strlen("Can't return attributes."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "Can't return attributes.", nb); return STRS_ERROR; } for (ip=0; ip<propList->nProps, ip++) { if (strcmp("A",propList->vProps[ip].name)==0) { /* Variable "a" is declared as a * character string, and typically * contains a value set by APP_Configure. */ if (a == NULL strlen(a) == 0) { propList->vProps[ip].value = NULL; } else { propList->vProps[ip].value = a; } } } return STRS_OK; } </pre>

NASA-STD-4009

Table 10—APP_Read()

APP_Read()	
Description	Method used to obtain data from the application. This is optional. The API is defined in STRS_Source. The caller manages the buffer area, preallocating the buffer before calling APP_Read and processing the returned data without any effects on the data source application.
Parameters	<ul style="list-style-type: none"> • buffer - (out STRS_Message) a pointer to an area in which the application stores the requested data • nb - (in STRS_Buffer_Size) number of bytes requested
Return	Error status (negative) or actual number of bytes (non-negative) obtained (STRS_Result)
Precondition	The application is in the STRS_APP_RUNNING state. Storage for the buffer with space for nb bytes is allocated before calling APP_Read. If used for a C-style character string, the size should include space for a final '\0'.
Postcondition	The data from the application is stored in the buffer area.
See Also	STRS_Read
Example	<pre>STRS_Result APP_Read(STRS_Message buffer, STRS_Buffer_Size nb) { if (nb <= 4) return STRS_ERROR; strcpy (buffer, "ABCD"); return strlen(buffer); }</pre>

NASA-STD-4009

Table 11—APP_ReleaseObject()

APP_ReleaseObject()	
Description	Free any resources that the application has acquired. An example would be to close any open files or devices. Nothing is done if the application state is STRS_APP_RUNNING. The API is defined in STRS_LifeCycle. The method is similar to releaseObject() in LifeCycle interface in SCA or OMG/SWRADIO. The purpose of APP_ReleaseObject is to prepare the application for removal.
Parameters	None.
Return	status (STRS_Result)
Precondition	Application is in the STRS_APP_INSTANTIATED or STRS_APP_STOPPED state.
Postcondition	All resources acquired by the application are released.
See Also	STRS_ReleaseObject
Example	<pre> STRS_Result APP_ReleaseObject() { if (myState == STRS_APP_RUNNING) { STRS_Buffer_Size nb = strlen("Can't free resources when RUNNING."); STRS_Log(fromWF, STRS_WARNING_QUEUE, "Can't free resources when RUNNING.", nb); return STRS_WARNING; } else { ... } return STRS_OK; } </pre>

NASA-STD-4009

Table 12—APP_RunTest()

APP_RunTest()	
Description	Test specific functionality within the application. The tests provide aid in isolating faults within the application. Application may be in any state, but certain tests may be restricted to specific states. Property values may be specified or retrieved. The propList may be NULL if it is not used. The caller manages the propList, preallocating the structure. The caller fills in the appropriate list of names and any input values and sets nProps to the number of names in the list (nProps > 0). The application fills in any output values for those Properties whose names are specified in the propList. The API is defined in STRS_TestableObject. The method is similar to runTest() in TestableObject interface in SCA and OMG/SWRADIO.
Parameters	<ul style="list-style-type: none"> testID - (in STRS_TestID) number of the test to be performed. A value of STRS_TEST_STATUS is always to be implemented to return to the current application state as shown in figure 15. Other values of testID are mission dependent. propList - (inout STRS_Properties*) list of the name and value pairs used to configure the test and/or return results.
Return	status or state (STRS_Result)
Precondition	The propList is to have space allotted for the maximum number of properties whose values are to be used or returned.
Postcondition	The test is performed. The state is unchanged unless specifically required by mission.
See Also	STRS_RunTest
Example	<pre> STRS_Result APP_RunTest(STRS testID, STRS_Properties *propList) { if (testID == STRS_TEST_STATUS) return myState; if (testID == STRS_TEST_USER_BASE) { ... } else { STRS_Buffer_Size nb = strlen("Invalid APP_RunTest argument test ID."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "Invalid APP_RunTest argument testID.",nb); } return STRS_ERROR; } </pre>

NASA-STD-4009

Table 13—APP_Start()

APP_Start()	
Description	Begin normal application processing. Nothing is done if the application is not in STRS_APP_STOPPED state. The API is defined in STRS_ControllableComponent. The method is similar to start() in the Resource interface in the SCA or ControllableComponent interface in the OMG/SWRADIO. If no fault is detected, this method changes the state to the STRS_APP_RUNNING state.
Parameters	None.
Return	status (STRS_Result)
Precondition	Application is in the STRS_APP_STOPPED state.
Postcondition	Application is in the STRS_APP_RUNNING state.
See Also	STRS_Start
Example	<pre>STRS_Result APP_Start() { if (myState == STRS_APP_STOPPED) { ... myState = STRS_APP_RUNNING; ... } else { return STRS_ERROR; } return STRS_OK; }</pre>

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Table 14—APP_Stop()

APP_Stop()	
Description	End normal application processing. Nothing is done unless the application is in STRS_APP_RUNNING state. The API is defined in STRS_ControllableComponent. The method is similar to stop() in the Resource interface in the SCA or ControllableComponent interface in the OMG/SWRADIO. If no fault is detected, this method changes the state to the STRS_APP_STOPPED state.
Parameters	None.
Return	status (STRS_Result)
Precondition	Application is in the STRS_APP_RUNNING state.
Postcondition	Application is in the STRS_APP_STOPPED state.
See Also	STRS_Stop
Example	<pre> STRS_Result APP_Stop() { if (myState == STRS_APP_RUNNING) { ... myState = STRS_APP_STOPPED; ... } else { return STRS_ERROR; } return STRS_OK; } </pre>

NASA-STD-4009

Table 15—APP_Write()

APP_Write()	
Description	Method used to send data to the application. This is optional. The API is defined in STRS_Sink. The caller manages the buffer area, preallocating and filling the buffer before calling APP_Write.
Parameters	<ul style="list-style-type: none"> • buffer - (in STRS_Message) pointer to the data for the application to process • nb - (in STRS_Buffer_Size) number of bytes in buffer
Return	Error status (negative) or number of bytes (non-negative) written (STRS_Result)
Precondition	Application is in the STRS_APP_RUNNING state. Storage for the buffer with space for nb bytes is allocated before calling APP_Write. If used for a C-style character string, the size should include space for a final '\0'.
Postcondition	The data has been captured by the application for its processing.
See Also	STRS_Write
Example	<pre>STRS_Result APP_Write(STRS_Message buffer, STRS_Buffer_Size nb) { /* Data in buffer is character data. */ if (strlen(buffer) != nb -1) return STRS_ERROR; int nco = fprintf(stdout,"%s\n",buffer); return (STRS_Result) nco; }</pre>

(STRS-29) Each STRS application shall contain a callable APP_Configure method as described in table 5, APP_Configure().

(STRS-30) Each STRS application shall contain a callable APP_GroundTest method as described in table 6, APP_GroundTest().

(STRS-31) Each STRS application shall contain a callable APP_Initialize method as described in table 7, APP_Initialize().

(STRS-32) Each STRS application shall contain a callable APP_Instance method as described in table 8, APP_Instance().

(STRS-33) Each STRS application shall contain a callable APP_Query method as described in table 9, APP_Query().

(STRS-34) If the STRS application provides data to the infrastructure, then the STRS application shall contain a callable APP_Read method as described in table 10, APP_Read().

(STRS-35) Each STRS application shall contain a callable APP_ReleaseObject method as described in table 11, APP_ReleaseObject().

(STRS-36) Each STRS application shall contain a callable APP_RunTest method as described in table 12, APP_RunTest().

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

(STRS-37) Each STRS application shall contain a callable APP_Start method as described in table 13, APP_Start().

(STRS-38) Each STRS application shall contain a callable APP_Stop method as described in table 14, APP_Stop().

(STRS-39) If the STRS application receives data from the infrastructure, then the STRS application shall contain a callable APP_Write method as described in table 15, APP_Write().

7.3.2 STRS Infrastructure-Provided Application Control API

The STRS infrastructure provides the STRS Infrastructure-provided Application Control API to support application operation using the STRS Application-provided Application Control API in section 7.3.1. These STRS Infrastructure-provided Application Control API methods (section 7.3.2 beginning with “STRS_” correspond to the STRS Application-provided Application Control API (section 7.3.1) beginning with “APP_”, and are used to access those STRS Application-provided Application Control API methods. The STRS infrastructure implements these STRS Infrastructure-provided Application Control API methods for use by any STRS application, or any part of the infrastructure that is desired to be implemented in a portable way.

A property structure contains a list of the name and value pairs used to set or get execution parameters (section 7.3.10).

NASA-STD-4009

Table 16—STRS_Configure()

STRS_Configure()	
Description	Set values for one or more properties in the target component (application, device). It is the responsibility of the target component to determine which properties can be changed in which states. The caller manages the propList, preallocating and filling in the names and values before calling STRS_Configure.
Parameters	<ul style="list-style-type: none"> • fromWF - (in STRS_HandleID) handle ID of current component making the request. • toWF - (in STRS_HandleID) handle ID of target component that should respond to the request. • propList - (in STRS_Properties *) list of name and value pairs.
Return	status (STRS_Result)
Precondition	Storage for the propList with space for sufficient name and value pairs; sufficient space for each name and value is allocated before calling STRS_Configure.
Postcondition	The appropriate named values are configured. The state is unchanged unless specifically required by the mission.
See Also	APP_Configure
Example	<pre> /* Set A=5, B=27. */ struct { STRS_NumberOfProperties nProps; STRS_NumberOfProperties mProps; STRS_Property vProps[MAX_PROPS]; } propList; propList.nProps = 2; propList.mProps = MAX_PROPS; propList.vProps[0].name = "A"; propList.vProps[0].value = "5"; propList.vProps[1].name = "B"; propList.vProps[1].value = "27"; STRS_Result rtn = STRS_Configure(fromWF,toWF, (STRS_Properties *) &propList); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("STRS_Configure fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "STRS_Configure fails.", nb); } </pre>

NASA-STD-4009

Table 17—STRS_GroundTest()

STRS_GroundTest()	
Description	Perform unit and system testing—usually done before deployment. The testing may include calibration. The tests aid in isolating faults within the target component. This method provides the exhaustive testing that is required before entrusting life and property to a software-defined radio. A responding application may be in any state, but certain tests may be restricted to specific states. Property values may be specified or retrieved. The propList may be NULL if it is not used. The caller manages the propList, preallocating the structure. The caller fills in the appropriate list of names and any input values and sets nProps to the number of names in the list (nProps > 0). The target component fills in any output values for those Properties whose names are specified in the propList. This method may be invalid upon deployment.
Parameters	<ul style="list-style-type: none"> • fromWF - (in STRS_HandleID) handle ID of current component making the request. • toWF - (in STRS_HandleID) handle ID of target component that should respond to the request. • testID - (in STRS_TestID) number of the test to be performed. Values are mission dependent. • propList - (inout STRS_Properties *) list of the name and value pairs used to configure the test and/or return results.
Return	status or state (STRS_Result)
Precondition	The propList is to have space allotted for the maximum number of properties whose values are to be used or returned.
Postcondition	The test is performed. The state is unchanged unless specifically required by mission.
See Also	APP_GroundTest
Example	<pre> STRS_Result rtn = STRS_GroundTest(fromWF,toWF,testID,NULL); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("GroundTest fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "GroundTest fails.", nb); } </pre>

NASA-STD-4009

Table 18—STRS_Initialize()

STRS_Initialize()	
Description	Initialize the target component (application, device). The purpose is to set or reset the component to a known initial state.
Parameters	<ul style="list-style-type: none"> • fromWF - (in STRS_HandleID) handle ID of current component making the request. • toWF - (in STRS_HandleID) handle ID of target component that should respond to the request.
Return	status (STRS_Result)
Precondition	Application is in STRS_APP_INSTANTIATED or STRS_APP_STOPPED state.
Postcondition	Application is in STRS_APP_STOPPED state.
See Also	APP_Initialize
Example	<pre>STRS_Result rtn = STRS_Initialize(fromWF,toWF); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("STRS_Initialize fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "STRS_Initialize fails.", nb); }</pre>

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Table 19—STRS_Query()

STRS_Query()	
Description	Obtain values for one or more properties in the target component (application, device). The caller manages the propList, preallocating the structure. The propList may not be NULL. If the caller fills in the appropriate list of names and sets nProps to the number of names in the list (nProps > 0), only those values will be returned whose names are specified in the propList. If the caller specifies no names in propList (nProps = 0), both names and values are filled in up to the maximum number allotted (mProps).
Parameters	<ul style="list-style-type: none"> • fromWF - (in STRS_HandleID) handle ID of current component making the request. • toWF - (in STRS_HandleID) handle ID of target component that should respond to the request. • propList - (inout STRS_Properties *) - list of name and value pairs
Return	status (STRS_Result)
Precondition	The propList is to have space allotted for the maximum number of properties encountered.
Postcondition	propList is populated with values if names are already in the list; otherwise, it is populated with all available names and values.
See Also	APP_Query
Example	<pre> struct { STRS_NumberOfProperties nProps; STRS_NumberOfProperties mProps; STRS_Property vProps[MAX_PROPS]; } propList; propList.nProps = 2; propList.mProps = MAX_PROPS; propList.vProps[0].name = "A"; propList.vProps[0].value = NULL; propList.vProps[1].name = "B"; propList.vProps[1].value = NULL; STRS_Result rtn = STRS_Query(fromWF, toWF, (STRS_Properties *) &propList); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("STRS_Query fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "STRS_Query fails.", nb); } for (ip=0; ip<propList.nProps; ip++) { cout << propList.vprops[ip].name << "=" << propList.vProps[ip].value << std::endl; } </pre>

NASA-STD-4009

Table 20—STRS_ReleaseObject()

STRS_ReleaseObject()	
Description	Free any resources that the target component (application, device) has acquired. An example would be to allow the target component to close any open files or devices. Nothing is done if the application is started. The purpose of STRS_ReleaseObject is to prepare the target component for removal.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. toWF - (in STRS_HandleID) handle ID of target component that should respond to the request.
Return	status (STRS_Result)
Precondition	Application is in STRS_APP_INSTANTIATED or STRS_APP_STOPPED state
Postcondition	All resources acquired by the application are released.
See Also	APP_ReleaseObject
Example	<pre> STRS_Result rtn = STRS_ReleaseObject(fromWF,toWF); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("STRS_ReleaseObject fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "STRS_ReleaseObject fails.", nb); } </pre>

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Table 21—STRS_RunTest()

STRS_RunTest()	
Description	Test specific functionality within the target component (application, device). The tests provide aid in isolating faults within the target component. A responding application may be in any state, but certain tests may be restricted to specific states. Property values may be specified or retrieved. The propList may be NULL if it is not used. The caller manages the propList, preallocating the structure. The caller fills in the appropriate list of names and any input values and sets nProps to the number of names in the list (nProps > 0). The target component fills in any output values for those Properties whose names are specified in the propList.
Parameters	<ul style="list-style-type: none"> • fromWF - (in STRS_HandleID) handle ID of current component making the request. • toWF - (in STRS_HandleID) handle ID of target component that should respond to the request. • testID - (in STRS_TestID) number of the test to be performed. A value of STRS_TEST_STATUS is always to be implemented to return the current target component state as shown in figure 15. Other values of testID are mission-dependent. • propList - (inout STRS_Properties*) list of the name and value pairs used to configure the test and/or return results.
Return	status or state (STRS_Result)
Precondition	The propList is to have space allotted for the maximum number of properties whose values are to be used or returned.
Postcondition	The test is performed. The state is unchanged unless specifically required by mission.
See Also	APP_RunTest
Example	<pre> STRS_Result state = STRS_RunTest(fromWF,toWF, STRS_TEST_STATUS,NULL); if (! STRS_IsOK(state)) { STRS_Buffer_Size nb = strlen("STRS_RunTest fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "STRS_RunTest fails.", nb); } </pre>

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Table 22—STRS_Start()

STRS_Start()	
Description	Begin target component (application, device) processing. Nothing is done if the application (or device) is already started.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. toWF - (in STRS_HandleID) handle ID of target component that should respond to the request.
Return	status (STRS_Result)
Precondition	Application is in the STRS_APP_STOPPED state.
Postcondition	Application is in the STRS_APP_RUNNING state.
See Also	APP_Start
Example	<pre>STRS_Result rtn = STRS_Start(fromWF,toWF); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("STRS_Start fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "STRS_Start fails.", nb); }</pre>

Table 23—STRS_Stop()

STRS_Stop()	
Description	End target component (application, device) processing. Nothing is done unless the application (or device) is started.
Parameters	<ul style="list-style-type: none"> fromWF - in STRS_HandleID) handle ID of current component making the request. toWF - in STRS_HandleID) handle ID of target component that should respond to the request.
Return	status (STRS_Result)
Precondition	Application is in the STRS_APP_RUNNING state.
Postcondition	Application is in the STRS_APP_STOPPED state.
See Also	APP_Stop
Example	<pre>STRS_Result rtn = STRS_Stop(fromWF,toWF); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("STRS_Stop fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "STRS_Stop fails.", nb); }</pre>

(STRS-40) The STRS infrastructure shall contain a callable STRS_Configure method as described in table 16, STRS_Configure().

(STRS-41) The STRS infrastructure shall contain a callable STRS_GroundTest method as described in table 17, STRS_GroundTest().

(STRS-42) The STRS infrastructure shall contain a callable STRS_Initialize method as described in table 18, STRS_Initialize().

(STRS-43) The STRS infrastructure shall contain a callable STRS_Query method as described in table 19, STRS_Query().

(STRS-44) The STRS infrastructure shall contain a callable STRS_ReleaseObject method as described in table 20, STRS_ReleaseObject().

(STRS-45) The STRS infrastructure shall contain a callable STRS_RunTest method as described in table 21, STRS_RunTest().

(STRS-46) The STRS infrastructure shall contain a callable STRS_Start method as described in table 22, STRS_Start().

(STRS-47) The STRS infrastructure shall contain a callable STRS_Stop method as described in table 23, STRS_Stop().

7.3.3 STRS Infrastructure Application Setup API

The STRS infrastructure Application Setup methods are general methods or are used to control one application from another.

NASA-STD-4009

Table 24—STRS_AbortApp()

STRS_AbortApp()	
Description	Abort an application or service.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. toWF - (in STRS_HandleID) handle ID of target component that should respond to the request
Return	Status (STRS_Result)
Precondition	Application is in the STRS_APP_INSTANTIATED, STRS_APP_STOPPED, or STRS_APP_RUNNING state.
Postcondition	The target component is aborted, and application is stopped, resources released, and unloaded, if allowed by OE.
See Also	N/A.
Example	<pre>STRS_Result rtn = STRS_AbortApp(fromWF,toWF); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("AbortApp fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "AbortApp fails.", nb); }</pre>

Table 25—STRS_GetErrorQueue()

STRS_GetErrorQueue()	
Description	Transform an error status into an error queue.
Parameters	<ul style="list-style-type: none"> result - (in STRS_Result) return value of previous call.
Return	Handle ID (STRS_HandleID) corresponding to invalid STRS_Result; that is, return STRS_ERROR_QUEUE for STRS_ERROR, STRS_WARNING_QUEUE for STRS_WARNING, and STRS_FATAL_QUEUE for STRS_FATAL.
Precondition	Any.
Postcondition	The corresponding error queue handle ID is returned.
See Also	STRS_IsOK
Example	<pre>char toWF[MAX_PATH_LENGTH]; strcpy(toWF, "/path/STRS_WFxxx.cfg"); STRS_HandleID wfID = STRS_InstantiateApp(fromWF,toWF); if (wfID < 0) { STRS_Buffer_Size nb = strlen("InstantiateApp fails."); STRS_Log(fromWF, STRS_GetErrorQueue((STRS_Result)wfID), "InstantiateApp fails.", nb); }</pre>

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Table 26—STRS_HandleRequest()

STRS_HandleRequest()	
Description	The table of object names is searched for the given name, and the index is returned as the handle ID. A handle ID is an identifier that is used to control access to applications and resources such as other applications, devices, files, or message queues. The handle ID of the current component (fromWF) is used for any error message unless the handle ID of the current component is what is being determined.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request unless it is a request for the handle ID of the current component. toWF - (in char *) name of desired resource (application, device, file, queue).
Return	Handle ID of the entity or error status. (STRS_HandleID)
Precondition	Any.
Postcondition	No change.
See Also	N/A.
Example	<pre> STRS_HandleID toWF = STRS_HandleRequest(fromWF, otherWF); if (toWF < 0) { STRS_Buffer_Size nb = strlen("Did not find handle."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "Did not find handle.", nb); }else { cout << "Found handle for " << otherWF << ": " << toWF << std::endl; } </pre>

NASA-STD-4009

Table 27—STRS_InstantiateApp()

STRS_InstantiateApp()	
Description	Instantiate an application, service, or device and perform any operations imposed by the configuration file. The configuration file specifies such items as initialization values and state. The infrastructure is responsible for calling the appropriate methods (e.g., STRS_Configure and/or APP_Configure) to configure the initial or default values. Other STRS methods may be called to perform additional functions, such as loading images or performing change of state as described in the application state diagram, figure 15.
Parameters	<ul style="list-style-type: none"> • fromWF - (in STRS_HandleID) handle ID of current component making the request. • toWF - (in char *) storage area name or fully qualified file name of the deployed configuration file of the application (or device) that should be instantiated. The handleName corresponding to the application, service, or device specified in the configuration file is to be unique. The convention is to prefix the application name with a unique source and add a number at the end, if required, to make the handleName unique. See section 9 for more information about configuration file(s).
Return	Handle ID (STRS_HandleID) of the application instantiated or the error status
Precondition	The files for the STRS application is to be accessible.
Postcondition	Application, service, or device is in the STRS_APP_INSTANTIATED state unless otherwise specified by the configuration file.
See Also	N/A.
Example	<pre> char toWF[MAX_PATH_LENGTH]; strcpy(toWF, "/path/STRS_WFxxx.cfg"); STRS_HandleID wfID = STRS_InstantiateApp(fromWF, toWF); if (wfID < 0) { STRS_Buffer_Size nb = strlen("InstantiateApp fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "InstantiateApp fails.", nb); } </pre>

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Table 28—STRS_IsOK()

STRS_IsOK()	
Description	Return true, if return value of previous call is not an error status.
Parameters	<ul style="list-style-type: none"> result - (in STRS_Result) return value of previous call.
Return	true, if STRS_Result is not STRS_WARNING, STRS_ERROR, or STRS_FATAL; that is, non-negative (bool)
Precondition	Previous call returns a status result.
Postcondition	No change.
See Also	STRS_GetErrorQueue
Example	<pre>char toWF[MAX_PATH_LENGTH]; strcpy(toWF, "/path/STRS_WFxxx.cfg"); STRS_HandleID wfID = STRS_InstantiateApp(fromWF, toWF); if (! STRS_IsOK((STRS_Result)wfID)) { STRS_Buffer_Size nb = strlen("InstantiateApp fails."); STRS_Log(fromWF, STRS_GetErrorQueue(wfID), "InstantiateApp fails.", nb); }</pre>

Table 29—STRS_Log()

STRS_Log()	
Description	Send log message for distribution as appropriate. The time stamp and an indication of the from and target handles are added automatically. STRS_Log may be used to inform the infrastructure that the STRS component is in the FAULT state when a target handle ID of STRS_ERROR_QUEUE, STRS_WARNING_QUEUE, or STRS_FATAL_QUEUE is used.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. logTarget - (in STRS_HandleID) handle ID of target (e.g., STRS_TELEMETRY_QUEUE, STRS_ERROR_QUEUE, STRS_WARNING_QUEUE, or STRS_FATAL_QUEUE). The last three special-purpose handle IDs may be used to log errors. msg - (in STRS_Message) a pointer to the data to process nb - (in STRS_Buffer_Size) number of bytes in buffer
Return	status (STRS_Result)
Precondition	The target queue component is in the STRS_APP_RUNNING state.
Postcondition	Log message is distributed.
See Also	See STRS_RunTest or APP_RunTest for further examples.
Example	<pre>STRS_Buffer_Size nb = strlen("file does not exist."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "file does not exist.", nb); // This could produce a line something like: // 19700101000000;WF1,ERROR,file does not exist.</pre>

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

(STRS-48) The STRS infrastructure shall contain a callable STRS_AbortApp method as described in table 24, STRS_AbortApp().

(STRS-49) The STRS infrastructure shall contain a callable STRS_GetErrorQueue method as described in table 25, STRS_GetErrorQueue().

(STRS-50) The STRS infrastructure shall contain a callable STRS_HandleRequest method as described in table 26, STRS_HandleRequest().

(STRS-51) The STRS infrastructure shall contain a callable STRS_InstantiateApp method as described in table 27, STRS_InstantiateApp().

(STRS-52) The STRS infrastructure shall contain a callable STRS_IsOK method as described in table 28, STRS_IsOK().

(STRS-53) The STRS infrastructure shall contain a callable STRS_Log method as described in table 29, STRS_Log().

(STRS-54) When an STRS application has a nonfatal error, the STRS application shall use the callable STRS_Log method as described in table 29, STRS_Log(), with a target handle ID of constant STRS_ERROR_QUEUE.

(STRS-55) When an STRS application has a fatal error, the STRS application shall use the callable STRS_Log method as described in table 29, STRS_Log(), with a target handle ID of constant STRS_FATAL_QUEUE.

(STRS-56) When an STRS application has a warning condition, the STRS application shall use callable the STRS_Log method as described in table 29, STRS_Log(), with a target handle ID of constant STRS_WARNING_QUEUE.

(STRS-57) When an STRS application needs to send telemetry, the STRS application shall use the callable STRS_Log method as described in table 29, STRS_Log(), with a target handle ID of constant STRS_TELEMETRY_QUEUE.

7.3.4 STRS Infrastructure Data Sink

The STRS Infrastructure Data Sink method, STRS_Write, is used to push data to any implemented data sink. A data sink may be an STRS application or STRS Device implementing APP_Write, a queue, or a file opened for writing.

(STRS-58) The STRS infrastructure shall contain a callable STRS_Write method as described in table 30, STRS_Write().

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

Table 30—STRS_Write()

STRS_Write()	
Description	Method used to send data to a target component (application, device, file, or queue) acting as a sink. The caller manages the buffer area, preallocating and filling the buffer before calling STRS_Write.
Parameters	<ul style="list-style-type: none"> • fromWF - (in STRS_HandleID) handle ID of current component making the request. • toID - (in STRS_HandleID) handle ID of target component that should respond to the request and that implemented STRS_Sink. • buffer - (in STRS_Message) a pointer to the data to process • nb - (in STRS_Buffer_Size) number of bytes in buffer
Return	Error status (negative) or number of bytes (non-negative) written (STRS_Result)
Precondition	The target component is in the STRS_APP_RUNNING state. Storage for the buffer is allocated before calling STRS_Write having space for at least nb bytes. If used for a C-style character string, the size should include space for a final '\0'.
Postcondition	The data has been captured by the target component for its processing.
See Also	APP_Write
Example	<pre>char buffer[32]; strcpy(buffer, "ABCDE"); STRS_Buffer_Size nb = strlen(buffer); STRS_Result rtn = STRS_Write(fromWF, toID, buffer, nb);</pre>

7.3.5 STRS Infrastructure Data Source

The STRS Infrastructure Data Source method, STRS_Read, is used to pull data from any implemented data source or supplier. A data source may be an STRS application or STRS Device implementing APP_Read, a SIMPLE queue, or a file opened for reading.

(STRS-59) The STRS infrastructure shall contain a callable STRS_Read method as described in table 31, STRS_Read().

Table 31—STRS_Read()

STRS_Read()	
Description	Method used to obtain data from a target component (application, device, file, or SIMPLE queue) acting as a source or supplier. The caller manages the buffer area, preallocating the buffer before calling STRS_Read and processing the returned data without any effects on the data source application.
Parameters	<ul style="list-style-type: none"> • fromWF - (in STRS_HandleID) handle ID of current component making the request. • pullID - (in STRS_HandleID) handle ID of target component that should respond to the request and that implemented STRS_Source. • buffer - (out STRS_Message) a pointer to an area in which to store the data requested • nb - (in STRS_Buffer_Size) number of bytes requested
Return	Error status (negative) or actual number of bytes (non-negative) obtained (STRS_Result)
Precondition	The target component is in the STRS_APP_RUNNING state. Storage for the buffer is allocated before calling STRS_Read, having space for at least nb bytes. If used for a C-style character string, the size should include space for a final '\0'.
Postcondition	The data from the target component is stored in the buffer area.
See Also	APP_Read
Example	<pre>char buffer[32]; STRS_Buffer_Size nb = 32; STRS_Result rtn = STRS_Read(fromWF,pullID,buffer,nb);</pre>

7.3.6 STRS Infrastructure Device Control API

An STRS Device is a proxy for the data and/or control path to the actual hardware. An STRS Device is a “bridge” used to “decouple an abstraction from its implementation so that the two can vary independently.” An STRS Device is called using the methods in the STRS infrastructure Device Control API (as described in the tables below), STRS Infrastructure-provided Application Control API, Infrastructure Data Source API (if appropriate), and Infrastructure Data Sink API (if appropriate) to control the STRS Devices. The STRS Device may be implemented using any available platform-specific HAL to communicate with and control the specialized hardware. An STRS Device may also be used to hide the details of networking from the application. The purpose of abstracting the hardware interfaces in a standard manner is to make the applications more portable. An STRS Device is an STRS application that responds to the STRS Infrastructure-provided Application Control API (section 7.3.2) calls, the STRS Infrastructure Data Source API (section 7.3.5) calls (if appropriate), and STRS Infrastructure Data Sink API (section 7.3.4) calls (if appropriate), as well as the following additional calls. The STRS Device implementation is suggested in figure 14.

NASA-STD-4009

Table 32—STRS_DeviceClose()

STRS_DeviceClose()	
Description	Close the device.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. toDev - (in STRS_HandleID) handle ID of device that should respond to the request.
Return	status (STRS_Result)
Precondition	The device is open.
Postcondition	The device is closed.
See Also	N/A.
Example	<pre> STRS_Result rtn = STRS_DeviceClose(fromWF,toDev); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("DeviceClose fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "DeviceClose fails.", nb); } </pre>

Table 33—STRS_DeviceFlush()

STRS_DeviceFlush()	
Description	Send any buffered data immediately to the underlying hardware and clear the buffers.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. toDev - (in STRS_HandleID) handle ID of device that should respond to the request.
Return	status (STRS_Result)
Precondition	The device is open.
Postcondition	The device's buffered data is flushed.
See Also	N/A.
Example	<pre> STRS_Result rtn = STRS_DeviceFlush(fromWF,toDev); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("DeviceFlush fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "DeviceFlush fails.", nb); } </pre>

Table 34—STRS_DeviceLoad()

STRS_DeviceLoad()	
Description	Load a binary image to the device.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. toDev - (in STRS_HandleID) handle ID of device that should respond to the request. fileName - (in char *) storage area name or fully qualified file name of the binary image to load onto the hardware device.
Return	status (STRS_Result)
Precondition	The target device is open.
Postcondition	The binary image is stored in the target device.
See Also	N/A.
Example	<pre> STRS_Result rtn = STRS_DeviceLoad(fromWF,toDev, "/path/WF1.FPGA.bit"); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("DeviceLoad fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "DeviceLoad fails.", nb); } </pre>

Table 35—STRS_DeviceOpen()

STRS_DeviceOpen()	
Description	Open the device.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. toDev - (in STRS_HandleID) handle ID of device that should respond to the request.
Return	status (STRS_Result)
Precondition	The device is not already open.
Postcondition	The device is opened.
See Also	N/A.
Example	<pre> STRS_Result rtn = STRS_DeviceOpen(fromWF,toDev); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("DeviceOpen fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "DeviceOpen fails.", nb); } </pre>

NASA-STD-4009

Table 36—STRS_DeviceReset()

STRS_DeviceReset()	
Description	Reinitialize the device. Reset is normally used after the corresponding device has been started and stopped, and before the device is started again.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. toDev - (in STRS_HandleID) handle ID of device that should respond to the request.
Return	status (STRS_Result)
Precondition	The device is open.
Postcondition	The device is reset to an initial state.
See Also	N/A,
Example	<pre> STRS_Result rtn = STRS_DeviceReset(fromWF,toDev); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("DeviceReset fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "DeviceReset fails.", nb); } </pre>

Table 37—STRS_DeviceStart()

STRS_DeviceStart()	
Description	Start the device. This is normally not used since most devices start when they are loaded.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. toDev - (in STRS_HandleID) handle ID of device that should respond to the request.
Return	status (STRS_Result)
Precondition	The device is in the STRS_APP_STOPPED state.
Postcondition	The device is in the STRS_APP_RUNNING state.
See Also	N/A.
Example	<pre> STRS_Result rtn = STRS_DeviceStart(fromWF,toDev); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("DeviceStart fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "DeviceStart fails.", nb); } </pre>

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Table 38—STRS_DeviceStop()

STRS_DeviceStop()	
Description	Stop the device. This is normally not used since most devices stop when they are unloaded or when there are no data to process.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. toDev - (in STRS_HandleID) handle ID of device that should respond to the request.
Return	status (STRS_Result)
Precondition	The device is in the STRS_APP_RUNNING state.
Postcondition	The device is in the STRS_APP_STOPPED state.
See Also	N/A.
Example	<pre> STRS_Result rtn = STRS_DeviceStop(fromWF,toDev); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("DeviceStop fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "DeviceStop fails.", nb); } </pre>

Table 39—STRS_DeviceUnload()

STRS_DeviceUnload()	
Description	Unload the device.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. toDev - (in STRS_HandleID) handle ID of device that should respond to the request.
Return	status (STRS_Result)
Precondition	The device is loaded.
Postcondition	The device is unloaded.
See Also	N/A.
Example	<pre> STRS_Result rtn = STRS_DeviceUnload(fromWF,toDev); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("DeviceUnload fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "DeviceUnload fails.", nb); } </pre>

Table 40—STRS_SetISR()

STRS_SetISR()	
Description	Set the Interrupt Service Routine for the device.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of the current component making the request. toDev - (in STRS_HandleID) handle ID of the device that should respond to the request. pfun – (in STRS_ISR_Function) function pointer to a static function with no arguments to be called to service the interrupt
Return	status (STRS_Result)
Precondition	Any.
Postcondition	ISR function is activated.
See Also	N/A.
Example	<pre> qnew=myQ; fp = (STRS_ISR_Function) Test_ISR_Method; fprintf(stdout,"Pointer to function Test_ISR_Method: %p\n",fp); rtn = STRS_SetISR(myQ,qnew,(STRS_ISR_Function) fp); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("STRS_SetISR fails for Test_ISR_Method."); STRS_Log(myQ, STRS_ERROR_QUEUE, "STRS_SetISR fails for Test_ISR_ Method.",nb); } </pre>

(STRS-60) The STRS applications shall use the methods in the STRS infrastructure Device Control API, STRS Infrastructure-provided Application Control API, Infrastructure Data Source API (if appropriate), and Infrastructure Data Sink API (if appropriate) to control the STRS Devices.

(STRS-61) The STRS infrastructure shall contain a callable STRS_DeviceClose method as described in table 32, STRS_DeviceClose().

(STRS-62) The STRS infrastructure shall contain a callable STRS_DeviceFlush method as described in table 33, STRS_DeviceFlush().

(STRS-63) The STRS infrastructure shall contain a callable STRS_DeviceLoad method as described in table 34, STRS_DeviceLoad().

(STRS-64) The STRS infrastructure shall contain a callable STRS_DeviceOpen method as described in table 35, STRS_DeviceOpen().

(STRS-65) The STRS infrastructure shall contain a callable STRS_DeviceReset method as described in table 36, STRS_DeviceReset().

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

(STRS-66) The STRS infrastructure shall contain a callable STRS_ DeviceStart method as described in table 37, STRS_DeviceStart().

(STRS-67) The STRS infrastructure shall contain a callable STRS_ DeviceStop method as described in table 38, STRS_DeviceStop().

(STRS-68) The STRS infrastructure shall contain a callable STRS_DeviceUnload method as described in table 39, STRS_DeviceUnload().

(STRS-69) The STRS infrastructure shall contain a callable STRS_SetISR method as described in table 40, STRS_SetISR().

7.3.7 STRS Infrastructure File Control API

The STRS Infrastructure File Control methods, along with STRS_Read and/or STRS_Write, provide a portable means for the applications to use storage, the duration of which is mission-dependent. The word “file” is used to mean a named storage area regardless of the existence of a file system. The file control methods in POSIX PSE51 are not sufficient for the needs of STRS because an application strictly conforming to PSE51 can use the open(), fopen(), or freopen() functions only to open existing files, not to create new files. In addition, the PSE51 profile lacks functions to remove files or to provide information regarding available storage. For more information about POSIX, see section 7.4. The STRS Infrastructure File Control methods use a handle ID to access storage.

Table 41—STRS_FileClose()

STRS_FileClose()	
Description	Close the file. STRS_FileClose is used to close a file that has been opened by STRS_FileOpen.
Parameters	<ul style="list-style-type: none">fromWF - (in STRS_HandleID) handle ID of current component making the request.toFile - (in STRS_HandleID) handle ID of file to be closed.
Return	status (STRS_Result)
Precondition	The file is open.
Postcondition	The file is closed and the handle ID is released.
See Also	STRS_FileOpen
Example	<pre>STRS_Result rtn = STRS_FileClose(fromWF,toFile); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("FileClose fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "FileClose fails.", nb); }</pre>

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Table 42—STRS_FileGetFreeSpace()

STRS_FileGetFreeSpace()	
Description	Get total size of free space available for file storage.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. fileSystem - (in char *) used when more than one file system exists.
Return	Total size in bytes (STRS_File_Size).
Precondition	Any.
Postcondition	No change.
See Also	N/A.
Example	<pre>STRS_File_Size size = STRS_FileGetFreeSpace(fromWF, NULL); if (size < 0) { STRS_Buffer_Size nb = strlen("FileGetFreeSpace fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "FileGetFreeSpace fails.", nb); }</pre>

Table 43—STRS_FileGetSize()

STRS_FileGetSize()	
Description	Get the size of the specified file.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. fileName - (in char *) storage area name or fully qualified file name of the file for which the size is obtained.
Return	File size in bytes (STRS_File_Size).
Precondition	Any.
Postcondition	No change.
See Also	N/A.
Example	<pre>STRS_File_Size size = STRS_FileGetSize(fromWF, "/path/WF1.FPGA.bit"); if (size < 0) { STRS_Buffer_Size nb = strlen("FileGetSize fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "FileGetSize fails.", nb); }</pre>

Table 44—STRS_FileGetStreamPointer()

STRS_FileGetStreamPointer()	
Description	Get the file stream pointer for the file associated with the STRS handle ID. This is normally not used because either the common functions are built into the STRS architecture or the entire file manipulation is local to one application or device. This method may be required for certain file operations not built into the STRS architecture and distributed over more than one application or device or the STRS infrastructure. For example, the file stream pointer may be required when multiple applications write to the same file using a queue or need features not found in STRS_Write. Having a file system is optional; if no file system is present, NULL will be returned. A NULL will also be returned if another error condition is detected.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. toFile - (in STRS_HandleID) file handle ID.
Return	File stream pointer (FILE *) or NULL for error condition.
Precondition	File is open.
Postcondition	No change.
See Also	STRS_FileOpen
Example	<pre>FILE *fsp = STRS_FileGetStreamPointer(fromWF,toFile); if (fsp == NULL) { STRS_Buffer_Size nb = strlen("FileGetStreamPointer fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "FileGetStreamPointer fails.", nb); } else { rewind(fsp); }</pre>

Table 45—STRS_FileOpen()

STRS_FileOpen()	
Description	Open the file. This method is used to obtain an STRS handle ID when the file manipulation is either built into the STRS architecture or distributed over more than one application or device or the STRS infrastructure
Parameters	<ul style="list-style-type: none"> • fromWF - (in STRS_HandleID) handle ID of current component making the request. • filename - (in char *) file name of the file to be opened. • file access - (in STRS_Access) indicates if file is to be opened for reading, writing, both, or appending. • file type - (in STRS_Type) indicator whether file is text or binary.
Return	a handle ID used to read or write data from or to the file (STRS_HandleID)
Precondition	The file is not open.
Postcondition	The file is open unless an error occurs.
See Also	N/A.
Example	<pre> STRS_HandleID frd = STRS_FileOpen(fromWF, filename, STRS_ACCESS_READ, STRS_TYPE_TEXT); if (frd < 0) { STRS_Buffer_Size nb = strlen("FileOpen fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "FileOpen fails.", nb); } </pre>

NASA-STD-4009

Table 46—STRS_FileRemove()

STRS_FileRemove()	
Description	Remove the file.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. oldName - (in char *) name of file to be removed.
Return	status (STRS_Result)
Precondition	The existing file is not open.
Postcondition	The file is no longer available, and the space where it was stored becomes available.
See Also	N/A.
Example	<pre>STRS_Result rtn = STRS_FileRemove(fromWF, oldName); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("FileRemove fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "FileRemove fails.", nb); }</pre>

Table 47—STRS_FileRename()

STRS_FileRename()	
Description	Rename the file.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. oldName - (in char *) current name of file. newName - (in char *) new name of file after rename.
Return	status (STRS_Result)
Precondition	The existing file is not open. The new file should not exist.
Postcondition	The contents of the old file are now associated with the new file name.
See Also	N/A.
Example	<pre>STRS_Result rtn = STRS_FileRename(fromWF, oldName, newName); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("FileRename fails."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "FileRename fails.", nb); }</pre>

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

(STRS-70) The STRS infrastructure shall contain a callable STRS_FileClose method as described in table 41, STRS_FileClose().

(STRS-71) The STRS infrastructure shall contain a callable STRS_FileGetFreeSpace method as described in table 42, STRS_STRS_FileGetFreeSpace().

(STRS-72) The STRS infrastructure shall contain a callable STRS_FileGetSize method as described in table 43, STRS_FileGetSize().

(STRS-73) The STRS infrastructure shall contain a callable STRS_FileGetStreamPointer method as described in table 44, STRS_FileGetStreamPointer().

(STRS-74) The STRS infrastructure shall contain a callable STRS_FileOpen method as described in table 45, STRS_FileOpen().

(STRS-75) The STRS infrastructure shall contain a callable STRS_FileRemove method as described in table 46, STRS_FileRemove().

(STRS-76) The STRS infrastructure shall contain a callable STRS_FileRename method as described in table 47, STRS_FileRename().

7.3.8 STRS Infrastructure Messaging API

The STRS applications use the STRS Infrastructure Messaging methods to establish queues to send messages between components using a single queue handle ID. The ability for applications, services, devices, or files to communicate with other STRS applications, services, devices, or files is crucial for the separation of radio functionality among independent asynchronous components. For example, the receive and transmit telecommunication functionality can be separated between two applications. Another example is when commands or log messages come from several independent sources and have to be merged appropriately. Some examples of independent components that probably need to interact with others could be for navigation, GPS, file upload, file download, and computations (even nonradio). The STRS radio is essentially a computer, and it has capabilities that make the whole spacecraft system more robust. The final destination of a message is not necessarily known to the producer of the message.

There are two models for passing messages: STRS_QUEUE_SIMPLE and STRS_QUEUE_PUBSUB. In an STRS_QUEUE_SIMPLE queue, messages are written to a queue by one application and read from the queue by another application. In an STRS_QUEUE_PUBSUB queue, messages written to the queue by one application are subsequently written to all subscribers of that queue. Therefore, the STRS_QUEUE_PUBSUB messaging API should be implemented using a form of the Observer or Publish-Subscribe design pattern. To read from a SIMPLE queue, STRS_Read is used. To write to a queue, STRS_Write is used. STRS_Read and STRS_Write, provide a portable means for the applications to use queues. Specific predefined queues for the handle IDs denoted by STRS_ERROR_QUEUE, STRS_FATAL_QUEUE, and STRS_WARNING_QUEUE are required. The STRS_Log method uses these special-purpose handle IDs to log errors.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

The queue names are global so that the queues with the same name refer to the same queue across all applications. The same handle name refers to the same application, device, file, queue, timer, or service across all applications. For information about errors see section 7.3.11.

Table 48—STRS_QueueCreate()

STRS_QueueCreate()	
Description	Create a queue (first in, first out—FIFO). The use of the queue priority parameter is implementation dependent.
Parameters	<ul style="list-style-type: none"> • fromWF - (in STRS_HandleID) handle ID of current component making the request. • queueName - (in char *) unique name of the queue • queueType - (in STRS_Queue_Type) type of queue created: STRS_QUEUE_SIMPLE or STRS_QUEUE_PUBSUB. • queuePriority - (in STRS_Priority) priority of queue: STRS_PRIORITY_LOW, STRS_PRIORITY_MEDIUM, or STRS_PRIORITY_HIGH.
Return	handle ID of queue or error status (STRS_HandleID)
Precondition	Queue does not already exist having the given queue name.
Postcondition	Queue is created.
See Also	N/A.
Example	<pre> STRS_HandleID qX = STRS_QueueCreate(myQ, "QX", STRS_QUEUE_SIMPLE, STRS_PRIORITY_MEDIUM); if (qX < 0) { STRS_Buffer_Size nb = strlen("Can't create queue."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "Can't create queue.", nb); return STRS_ERROR; } </pre>

NASA-STD-4009

Table 49—STRS_QueueDelete()

STRS_QueueDelete()	
Description	Delete a queue. Any association between a publisher and subscriber that references the queue to be deleted is removed.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. toQueue - (inout STRS_HandleID) handle ID of queue to delete; either publisher or subscriber
Return	status (STRS_Result)
Precondition	Queue already exists having the specified queue handle ID.
Postcondition	Queue is deleted.
See Also	N/A.
Example	<pre>STRS_Result rtn = STRS_QueueDelete(myQ, qX); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("Can't delete queue."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "Can't delete queue.", nb); }</pre>

Table 50—STRS_Register()

STRS_Register()	
Description	Register an association between a publisher and subscriber. Disallow adding an association such that the subscriber has another association back to the publisher because this would cause an infinite loop.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. useQID - (in STRS_HandleID) handle ID of queue of type STRS_QUEUE_PUBSUB that will be used in sink; the publisher. actQID - (in STRS_HandleID) handle ID of queue, file, device, or target component that should respond to the request; the subscriber.
Return	status (STRS_Result)
Precondition	The publisher queue of type STRS_QUEUE_PUBSUB exists.
Postcondition	Association between publisher and subscriber is registered, if allowed.
See Also	N/A.
Example	<pre>STRS_Result rtn = STRS_Register(myQ, qX, qFC); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("Can't register subscriber."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "Can't register subscriber.", nb); }</pre>

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

Table 51—STRS_Unregister()

STRS_Unregister()	
Description	Remove an association between a publisher and subscriber.
Parameters	<ul style="list-style-type: none"> fromWF - (in STRS_HandleID) handle ID of current component making the request. useQID - (in STRS_HandleID) handle ID of queue of type STRS_QUEUE_PUBSUB that was used in sink; the publisher. actQID - (in STRS_HandleID) handle ID of queue, file, device, or target component that should respond to the request; usually the subscriber.
Return	status (STRS_Result)
Precondition	The publisher queue of type STRS_QUEUE_PUBSUB exists.
Postcondition	Association between publisher and subscriber is removed.
See Also	N/A.
Example	<pre>STRS_Result rtn = STRS_Unregister(myQ, qX, qFC); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("Can't unregister subscriber."); STRS_Log(fromWF, STRS_ERROR_QUEUE, "Can't unregister subscriber.", nb); }</pre>

(STRS-77) The STRS applications shall use the STRS Infrastructure Messaging, STRS Infrastructure Data Source, and STRS Infrastructure Data Sink methods to establish queues to send messages between components.

(STRS-78) The STRS infrastructure shall contain a callable STRS_QueueCreate method as described in table 48, STRS_QueueCreate().

(STRS-79) The STRS infrastructure shall contain a callable STRS_QueueDelete method as described in table 49, STRS_QueueDelete().

(STRS-80) The STRS infrastructure shall contain a callable STRS_Register method as described in table 50, STRS_Register().

(STRS-81) The STRS infrastructure shall contain a callable STRS_Unregister method as described in table 51, STRS_Unregister().

7.3.9 STRS Infrastructure Time Control API

The STRS Infrastructure Time Control methods are used to access the hardware and software timers. If timers require synchronization with external clocks, a dedicated service should handle the communication required between the STRS radio and the external clock source, adjusting the time or offset for distance and velocity, before using these methods to adjust a corresponding internal timer. These methods also include conversion of time between seconds and nanoseconds, taken individually, and some implementation-specific object containing both.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Although nanoseconds are the units obtained by STRS_GetNanoseconds, that does not imply that the resolution is nanoseconds or that the underlying STRS_TimeWarp object contains its data in nanoseconds. For example, the underlying STRS_TimeWarp object could count ticks from some epoch and then STRS_GetSeconds and STRS_GetNanoseconds compute the seconds and nanoseconds from the same or a different epoch. These timers are expected to be used for relatively low accuracy timing such as time stamps, timed events, and time constraints. The timers are expected to be used for signal processing in the GPP if the GPP becomes fast enough.

Table 52—STRS_GetNanoseconds()

STRS_GetNanoseconds()	
Description	Get the number of nanoseconds from the STRS_TimeWarp object.
Parameters	<ul style="list-style-type: none"> twObj - (in STRS_TimeWarp) the STRS_TimeWarp object from which the nanoseconds portion of the time increment is extracted.
Return	Integer number of nanoseconds in the STRS_TimeWarp object representing a time interval. (STRS_int32)
Precondition	Any.
Postcondition	No change.
See Also	STRS_SetTimeWarp, STRS_GetSeconds
Example	<pre>STRS_TimeWarp base, timx; STRS_int32 nsec; STRS_Result rtn; STRS_Clock_Kind kx = 1; rtn = STRS_GetTime(fromWF, toDev, *base, kx, *timx); nsec = STRS_GetNanoseconds(base);</pre>

Table 53—STRS_GetSeconds()

STRS_GetSeconds()	
Description	Get the number of seconds from the STRS_TimeWarp object.
Parameters	<ul style="list-style-type: none"> twObj - (in STRS_TimeWarp) the STRS_TimeWarp object from which the nanoseconds portion of the time increment is extracted.
Return	integer number of seconds in the STRS_TimeWarp object representing a time interval. (STRS_int32)
Precondition	Any.
Postcondition	No change.
See Also	STRS_SetTimeWarp, STRS_GetNanoseconds
Example	<pre>STRS_TimeWarp base, timx; STRS_int32 isec; STRS_Result rtn; STRS_Clock_Kind kx = 1; rtn = STRS_GetTime(fromWF, toDev, *base, kx, *timx); isec = STRS_GetSeconds(base);</pre>

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

Table 54—STRS_GetTime()

STRS_GetTime()	
Description	Get the current base time and the corresponding time of a specified type (kind). The base clock/timer is usually a hardware timer. The variable kind is used to obtain a nonbase time at a specified offset from the base time. An offset is usually specified to ensure that the clock is monotonically increasing after a power reset or synchronized with another clock/timer. To compute the time interval between two nonbase times of different kinds, the function is called twice and the interval is modified by the difference between the two base times. An example of the difference between two nonbase times is shown in the example below.
Parameters	<ul style="list-style-type: none"> • fromWF - (in STRS_HandleID) handle ID of current component making the request. • toDev - (in STRS_HandleID) handle ID of device that should respond to the request. • baseTime - (inout STRS_TimeWarp) current time of the base timer. • kind - (in STRS_Clock_Kind) type of clock/timer. • kindTime - (inout STRS_TimeWarp) current time of the specified timer.
Return	status (STRS_Result)
Precondition	Any.
Postcondition	No change.
See Also	STRS_SetTime
Example	<pre> STRS_TimeWarp b1,b2,t1,t2,diff; STRS_int32 isec,nsec; STRS_Result rtn; STRS_Clock_Kind k1 = 1; STRS_Clock_Kind k2 = 2; rtn = STRS_GetTime(fromWF,toDev,*b1,k1,*t1); rtn = STRS_GetTime(fromWF,toDev,*b2,k2,*t2); /* The time difference between timer k1 and * timer k2 is computed by obtaining the two * times, t1 and t2, and adjusting for the * time difference between the two base times, * b2 and b1: */ isec = STRS_GetSeconds(t2) - (STRS_GetSeconds(t1) + (STRS_GetSeconds(b2) - STRS_GetSeconds(b1))); nsec = STRS_GetNanoseconds(t2) - (STRS_GetNanoseconds(t1) + (STRS_GetNanoseconds(b2) - STRS_GetNanoseconds(b1))); diff = STRS_GetTimeWarp(isec,nsec); </pre>

NASA-STD-4009

Table 55—STRS_GetTimeWarp()

STRS_GetTimeWarp()	
Description	Get the STRS_TimeWarp object containing the number of seconds and nanoseconds in the time interval.
Parameters	<ul style="list-style-type: none"> • isec - (in STRS_int32) number of seconds in the time interval • nsec - (in STRS_int32) number of nanoseconds in the fractional portion of the time interval
Return	STRS_TimeWarp object representing the time interval.
Precondition	Any.
Postcondition	No change.
See Also	STRS_GetNanoseconds, STRS_GetSeconds, STRS_SetTime
Example	<pre>STRS_TimeWarp delta; STRS_int32 isec = 1; /* Leap second. */ STRS_int32 nsec = 0; delta = STRS_GetTimeWarp(isec,nsec);</pre>

Table 56—STRS_SetTime()

STRS_SetTime()	
Description	Set the current time in the specified clock/timer by adjusting the time offset.
Parameters	<ul style="list-style-type: none"> • fromWF - (in STRS_HandleID) handle ID of current component making the request. • toDev - (in STRS_HandleID) handle ID of device that should respond to the request. • kind - (in STRS_Clock_Kind) type of clock/timer. • delta - (in STRS_TimeWarp) increment to add to specified clock/timer.
Return	status (STRS_Result)
Precondition	Any.
Postcondition	Time is adjusted.
See Also	STRS_GetTime
Example	<pre>STRS_TimeWarp delta; STRS_int32 isec = 1; /* Leap second */ STRS_int32 nsec = 0; STRS_Result rtn; STRS_Clock_Kind k1 = 1; delta = STRS_GetTimeWarp(isec,nsec); rtn = STRS_SetTime(fromWF,toDev,k1,delta);</pre>

NASA-STD-4009

Table 57—STRS_Synch()

STRS_Synch()	
Description	Synchronize clocks. The action depends on whether the clocks to be synchronized are internal or external.
Parameters	<ul style="list-style-type: none"> • fromWF - (in STRS_HandleID) handle ID of current component making the request. • refDev - (in STRS_HandleID) handle ID of reference device containing the reference clock/timer. • ref - (in STRS_Clock_Kind) type of reference clock/timer. • targetDev - (in STRS_HandleID) handle ID of target device to synchronize. • target - (in STRS_Clock_Kind) type of clock/timer to synchronize with reference clock/timer.
Return	status (STRS_Result)
Precondition	Any.
Postcondition	Clocks are synchronized.
See Also	N/A.
Example	<pre> qref = STRS_HandleRequest(myQ, "ReferenceClock"); iref = 0; qtgt = STRS_HandleRequest(myQ, "TargetClock"); itgt = 0; rtn = STRS_Synch(myQ, qref, iref, qtgt, itgt); if (! STRS_IsOK(rtn)) { STRS_Buffer_Size nb = strlen("STRS_Synch fails."); STRS_Log(myQ, STRS_ERROR_QUEUE, "STRS_Synch fails.", nb); } </pre>

(STRS-82) Any portion of the STRS Applications on the GPP needing time control shall use the STRS Infrastructure Time Control methods to access the hardware and software timers.

(STRS-83) The STRS infrastructure shall contain a callable STRS_GetNanoseconds method as described in table 52, STRS_GetNanoseconds().

(STRS-84) The STRS infrastructure shall contain a callable STRS_GetSeconds method as described in table 53, STRS_GetSeconds().

(STRS-85) The STRS infrastructure shall contain a callable STRS_GetTime method as described in table 54, STRS_GetTime().

(STRS-86) The STRS infrastructure shall contain a callable STRS_GetTimeWarp method as described in table 55, STRS_GetTimeWarp().

(STRS-87) The STRS infrastructure shall contain a callable STRS_SetTime method as described in table 56, STRS_SetTime().

(STRS-88) The STRS infrastructure shall contain a callable STRS_Synch method as described in table 57, STRS_Synch().

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

7.3.10 STRS Predefined Data

For portability, standard names are defined for various constants and data types, but the implementation of these definitions is mission dependent. The common symbols and data types defined to support the STRS infrastructure APIs are shown in table 58, STRS Predefined Data.

(STRS-89) The STRS platform provider shall provide an STRS.h file containing the STRS predefined data shown in table 58, STRS Predefined Data.

(STRS-106) An STRS application shall use the appropriate constant, typedef, or struct defined in table 58, STRS Predefined Data, when the data are used to interact with the STRS APIs.

Table 58—STRS Predefined Data

Typedefs	
	<ul style="list-style-type: none"> • STRS_Access - a type of number used to indicate how reading and/or writing of a file or queue is done. See also constants STRS_ACCESS_APPEND, STRS_ACCESS_BOTH, STRS_ACCESS_READ, and STRS_ACCESS_WRITE. • STRS_Buffer_Size – a type of number used to represent a buffer size in bytes. The type of the number is to be long enough to contain the maximum number of bytes to reserve or to transfer with a read or write. • STRS_Clock_Kind - a type of number used to represent a kind of clock or timer. The type of the number is to be long enough to contain the maximum number of kinds of clocks and timers. • STRS_File_Size - a type of number used to represent a size in bytes. The type of the number is to be long enough to contain the number of bytes in GPP storage. A negative value returned indicates an error. • STRS_HandleID - a type of number used to represent an STRS application, device, file, or queue. A negative value returned indicates an error. • STRS_int8 - an 8-bit signed integer • STRS_int16 - a 16-bit signed integer • STRS_int32 - a 32-bit signed integer • STRS_int64 - a 64-bit signed integer • STRS_ISR_Function - used to define static C-style function pointers passed to the STRS_SetISR() method. The function passed to the STRS_SetISR() method is defined with no arguments. • STRS_Message - a char array pointer used for messages. • STRS_NumberOfProperties - a type of number used to represent the number of properties in a Properties structure. • STRS_Queue_Type – a type of number used to represent the queue type. See also constants STRS_QUEUE_SIMPLE and STRS_QUEUE_PUBSUB. • STRS_Priority - a type of number used to represent the priority of a queue. See also constants STRS_PRIORITY_HIGH, STRS_PRIORITY_MEDIUM, STRS_PRIORITY_LOW. • STRS_Properties – shorthand for “struct Properties” • STRS_Property – shorthand for “struct Property” • STRS_Result - a type of number used to represent a return value, where negative indicates an error. • STRS_TestID – a type of number used to represent the built-in test or ground test to

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

	<p>be performed by APP_RunTest or APP_GroundTest, respectively. See also STRS_TEST_STATUS and STRS_TEST_USER_BASE.</p> <ul style="list-style-type: none"> • STRS_TimeWarp - a representation of a time delay. The values in the representation are to be able to hold the number of seconds and nanoseconds in the time delay so that the corresponding macros can extract them. The time delay is meant to be used for recurrent processes such as in health management. The implementation is mission and/or platform specific and is most likely a struct. The maximum number of seconds in a time delay cannot be greater than 2^{31} seconds (68 years). See also STRS_GetSeconds(), STRS_GetNanoseconds(), and STRS_GetTimeWarp(). • STRS_Type - a type of number used to indicate whether a file is text or binary. See also constants STRS_TYPE_BINARY and STRS_TYPE_TEXT. • STRS_uint8 - an 8-bit unsigned integer • STRS_uint16 - a 16-bit unsigned integer • STRS_uint32 - a 32-bit unsigned integer • STRS_uint64 - a 64-bit unsigned integer
Constants	<ul style="list-style-type: none"> • STRS_ACCESS_APPEND - writing is allowed such that previous data written are preserved and new data are written following any previous data. Corresponds to ISO C fopen mode "a". • STRS_ACCESS_BOTH - both reading and writing are allowed. Corresponds to ISO C fopen mode "r+" used for update. • STRS_ACCESS_READ - reading is allowed. Corresponds to ISO C fopen mode "r". • STRS_ACCESS_WRITE - writing is allowed. Corresponds to ISO C fopen mode "w". • STRS_OK - the STRS_Result is valid. See also STRS_IsOK(). • STRS_ERROR - the STRS_Result is invalid. This indicates an error such that the application or other component is still usable. Indicated by a negative value. See also STRS_IsOK() and STRS_GetErrorQueue(). • STRS_ERROR_QUEUE - the STRS_HandleID indicates that the log queue is for error messages. See also STRS_GetErrorQueue(). • STRS_FATAL - the STRS_Result is invalid. This indicates a serious error such that the application or other component is not usable. Indicated by a negative value. See also STRS_IsOK() and STRS_GetErrorQueue(). • STRS_FATAL_QUEUE - the STRS_HandleID indicates that the log queue is for fatal messages. The fatal queue is used for messages that the fault monitoring and recovery functions are to deal with immediately. The messages are sent to the Flight Computer for further handling. See also STRS_GetErrorQueue(). • STRS_PRIORITY_HIGH – a number representing a high-priority queue. • STRS_PRIORITY_MEDIUM – a number representing a medium-priority queue. • STRS_PRIORITY_LOW – a number representing a low-priority queue. • STRS_QUEUE_PUBSUB – a number representing a Publish/Subscribe queue type. • STRS_QUEUE_SIMPLE – a number representing a simple queue type. • STRS_TELEMETRY_QUEUE - the STRS_HandleID indicates that the log queue is for telemetry data. • STRS_TEST_STATUS – The numerical value of type STRS_TestID used as the argument to APP_RunTest and STRS_RunTest so that the state of the STRS application is returned. • STRS_TEST_USER_BASE – The numerical value of type STRS_TestID for the

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

	<p>lowest numbered user-defined test. Any STRS_TestID values lower than STRS_TEST_USER_BASE are reserved arguments to APP_RunTest. An example of a test type lower than STRS_TEST_USER_BASE is STRS_TEST_STATUS.</p> <ul style="list-style-type: none">• STRS_TYPE_BINARY - the value indicating that a file is a binary file.• STRS_TYPE_TEXT - the value indicating that a file is a text file.• STRS_WARNING - the STRS_Result is invalid. This indicates an error such that there may be little or no effect on the operation of the application or other component. Indicated by a negative value. See also STRS_IsOK() and STRS_GetErrorQueue().• STRS_WARNING_QUEUE - the STRS_HandleID indicates that the log queue is for warning messages. See also STRS_GetErrorQueue().• STRS_APP_FATAL - waveform, service, or device state indicating that a nonrecoverable error has occurred. See also STRS_GetErrorQueue().• STRS_APP_ERROR - waveform, service, or device state indicating that a recoverable error has occurred. See also STRS_GetErrorQueue().• STRS_APP_INSTANTIATED - waveform, service, or device state indicating that the object is instantiated and ready to accept messages.• STRS_APP_RUNNING - waveform, service, or device state indicating that STRS_Start() has been called.• STRS_APP_STOPPED - waveform, service, or device state indicating that STRS_Initialize() or STRS_Stop() has been called.
Structs	<ul style="list-style-type: none">• Property - a struct with two-character pointer variables: name and value. Using a structure allows treating a name and value pair as a single item.• Properties - a struct with two variables (nProps and mProps) of type STRS_NumberOfProperties, and an array of Property structures (vProps). The variable nProps contains the number of items in the vProps array. The variable mProps contains the maximum number of items in the vProps array. Using an array of structures allows treating each name and value pair as a single item in the vProps array.

7.3.11 Error Handling

Special-purpose handle IDs for errors include the following: STRS_ERROR_QUEUE, STRS_WARNING_QUEUE, and STRS_FATAL_QUEUE. The STRS_Log method uses these special-purpose handle IDs to log errors. A nonfatal error is a correctable condition such that the application is usable when the error is corrected. This nonfatal error is denoted by the STRS return value of STRS_ERROR and is logged using the STRS handle ID of STRS_ERROR_QUEUE. A warning is an indication of an impending error that is correctable if action is taken. This warning is denoted by the STRS return value of STRS_WARNING and is logged using the STRS handle ID of STRS_WARNING_QUEUE. A fatal error is a condition where the application is subsequently not usable and a reboot or reload is often necessary. This fatal error is denoted by the STRS return value of STRS_FATAL and is logged using the STRS handle ID of STRS_FATAL_QUEUE.

7.4 Portable Operating System Interface

POSIX refers to a family of IEEE standards 1003.n that describe the fundamental services and functions necessary to provide a UNIX-like kernel interface to applications. POSIX itself is not an OS but is instead the guaranteed programming interfaces available to the application programmer.

POSIX specifies a set of OS interfaces and services. POSIX is not specifically bound to a specific OS, and has in fact been implemented on top of OS such as Digital Equipment Corporation's (DEC's) OpenVMS (Virtual Memory System) and Microsoft Windows NT. However, the creation of POSIX is closely coupled to the UNIX OS and its evolution. The goal was to create a standard set of interfaces that all of the UNIX flavors would support in order to facilitate software portability. Even though POSIX technically refers to the family of specifications, it is more commonly used to refer specifically to IEEE 1003.1, which is the core POSIX specification.

Characteristics of POSIX include the following:

- a. Application-oriented.*
- b. Interface, not implementation.*
- c. Source, not object, portability.*
- d. The C-language/system interfaces written in terms of the ISO C standard.*
- e. No superuser, no system administration.*
- f. Minimal interface, minimally defined—core facilities of this Standard have been kept as minimal as possible.*
- g. Broadly implementable.*
- h. Minimal changes to historical implementations.*
- i. Minimal changes to existing application code.*

The original POSIX specification was based on a general-purpose computing platform, but a series of amendments addressed the unique requirements of real-time computing. These amendments follow:

- a. IEEE Std 1003.1b-1993 Realtime Extension.*
- b. IEEE Std 1003.1c-1995 Threads Extension.*
- c. IEEE Std 1003.1d-1999 Additional Realtime Extensions.*
- d. IEEE Std 1003.1j-2000 Advanced Realtime Extensions.*
- e. IEEE Std 1003.1q-2000 Tracing.*

These amendments were rolled into the base specification in version IEEE 1003.1-1996. IEEE 1003.13 provides a standards-based option for an STRS AEP.

7.4.1 STRS Application Environment Profile

The subset of the POSIX API described below is used by STRS applications to access platform services when no STRS Infrastructure-provided API is available. POSIX was chosen as part of this Standard because it defines an open-standard OS interface and environment to support application portability. However, because of the limited resources on a space-based platform, it was not practical to support the entire IEEE 1003.1 specification.

The POSIX 1003.1 standard provides a means to implement a subset of the interfaces by using “Subprofiling Option Groups.” These option groups specify “Units of Functionality” that can be removed from the base POSIX specification.

IEEE 1003.13 created four AEPs that specified subsets of 1003.1 more suitable to embedded applications. These profiles follow:

- PSE51—Minimal Realtime Systems Profile.*
- PSE52—Realtime Controller System Profile.*
- PSE53—Dedicated Realtime System Profile.*
- PSE54—Multi-Purpose Realtime System Profile.*

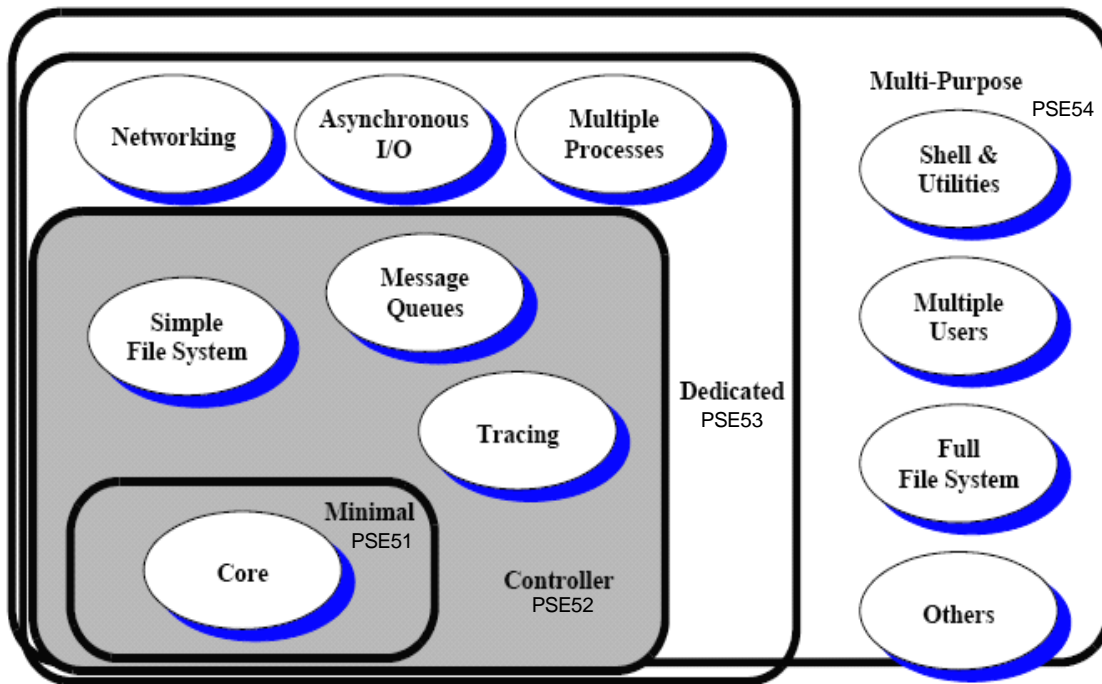


Figure 16—Profile Building Blocks

The profiles are each upwardly compatible and consist of the basic building blocks shown in figure 16,¹ Profile Building Blocks.

Each of these profiles has increasing capabilities, which increase requirements on resources. Profiles 51 and 52 run on a single processor with no Memory Management Unit (MMU), and thus imply a single process containing one or more threads. Profile 52 adds a file system interface and asynchronous I/O. Profile 53 adds support for multiple processes, thus requiring an MMU. The last and largest profile 54 adds support for interactive users, and is almost a full-blown POSIX 1003.1 environment. The higher numbered profiles are supersets of the lower numbered profiles, such that PSE52 includes all the features of a PSE51.

Upward portability between profiles is supported by requiring certain APIs, such as memory locking, for profiles PSE51 and PSE52. Even though there is no MMU support on the PSE51 and PSE52 profiles, code written as if there is an MMU present will be portable among all four profiles by requiring such APIs to be defined in all four profiles. The signature of these APIs will be identical on all profiles, but the functionality will differ according to the capabilities. For example, calling a memory-locking API on a PSE51 platform with no MMU will always return success. When this example application is ported to a PSE53 platform, the memory locking will work as intended without modification to the source code.

Currently this Standard supports platforms based on profiles PSE51 through PSE54, although PSE54 will only be used for development platforms and ground stations. Allowing multiple profiles allows the architecture to scale with mission class. Applications developed for a specific

¹ IEEE Std 1003.13-2003

NASA-STD-4009

profile are compatible with higher profiles; that is, a profile 52 application could be ported to profile PSE53 and PSE54 platform, but not vice versa. This upward scalability anticipates that smaller platforms will desire smaller profiles and will not have the resources to run larger applications that comply with the larger profiles. Appendix B provides a table comparing the POSIX profile functionality for subset PSE51 through PSE53.

(STRS-90) The STRS OE shall provide the interfaces described in POSIX IEEE Standard 1003.13-2003 profile PSE51.

For constrained-resource platforms with limited software evolutionary capability, where the waveform signal processing is implemented in specialized hardware, the supplier may request a waiver to only implement a subset of POSIX PSE51 as required by the portion of the waveforms residing on the GPP. The applications created for this platform are to be upward-compatible to a larger platform containing POSIX PSE51. The POSIX API is grouped into units of functionality. If none of the applications for a constrained-resource platform use any of the interfaces in a unit of functionality, then the supplier may request a waiver to eliminate that entire unit of functionality.

Regardless of the POSIX profile implemented, applications are not to use any restricted functions or their equivalent, such as abort(), atexit(), exit(), calloc(), free(), malloc(), or realloc(). For portability of application code to multithreaded radio platforms, STRS applications are to use the thread-safe versions of the POSIX methods listed in table 59, Replacements for Unsafe Functions.

(STRS-91) STRS applications shall use POSIX methods except for the unsafe functions listed in table 59, Replacements for Unsafe Functions.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

Table 59—Replacements for Unsafe Functions

Unsafe Function Do Not Use!	Reentrant Counterpart OK to Use
abort	STRS_AbortApp
asctime	asctime_r
atexit	-
calloc	-
ctermid	ctermid_r
ctime	ctime_r
exit	STRS_AbortApp
free	-
getlogin	getlogin_r
gmtime	gmtime_r
localtime	localtime_r
malloc	-
rand	rand_r
readdir	readdir_r
realloc	-
strtok	strtok_r
tmpnam	tmpnam_r

7.5 Network Stack

A network stack is the part of the OS used for networking, usually Transmission Control Protocol/Internet Protocol (TCP/IP). Communications over a network use a layered network model. TCP/IP is the protocol that is used to transport information over the internet, and the TCP/IP network model consists of five layers: the application layer, the transport layer, the network layer, the data link layer, and the physical network.

7.6 Operating System

The OS is an integral part of the OE for the STRS software architecture. Modern communication systems perform simultaneous application processing in dedicated hardware at the very fast speeds to which users have become accustomed. Any change in this environment is to equal or exceed previous performance for it to be considered for usage. As such, the proposal to perform application processing via software modules executing on a GPP requires careful consideration of both the necessary OS characteristics and the application processing requirements. In a simplistic sense, a computer OS manages the usage and sharing of resources between competing users (i.e., tasks) to perform work. In this case, each task is performing a specific instance of application processing. When the OS decides to stop the execution of one task and start another, the current context of the machine (register values, instruction pointers, etc.) is to be saved and then switched to accommodate the requirements of the new task. On a desktop computer system, context switching between competing tasks is performed on an ad-hoc basis with no guarantee of task execution. For most missions, this is unacceptable because context switching between execution threads and deterministic thread execution are the driving characteristics for an OS.

To support these requirements, most radio platforms will use an RTOS instead of a general-purpose OS. An RTOS provides the capabilities of fast, low overhead for context switching, and a deterministic scheduling mechanism so that processing constraints can be achieved when required.

Fundamental to STRS application development is the existence of an OS kernel that can be configured and scaled down to fit into the executable image of the STRS system. A modern RTOS is primarily designed for either performance (monolithic kernel) or extensibility (microkernel). Monolithic kernels have tightly integrated services and less run-time overhead but are not easily extensible. Microkernels have somewhat high run-time overheads but are highly extensible. Most modern RTOSs are microkernels, and although modern microkernels have more overhead than monolithic kernels, they have less overhead than traditional microkernels. The run-time overhead of modern RTOSs is decreased by reducing the unnecessary context switch. Important timings such as context switch time, interrupt latency, and semaphore get and release latency is to be kept to a minimum.

7.7 Hardware Abstraction Layer

The HAL is the library of software functions in the STRS OE that provides a platform-vendor-specific view of the specialized hardware by abstracting the underlying physical hardware interfaces. The HAL allows specialized hardware to be integrated with the GPM so that the STRS OE can access functions implemented on the specialized hardware of the STRS platform.

Two examples of specialized hardware currently in use on SDRs are FPGAs and DSPs. Examples of functionality that a HAL might need to support include boot code for initializing the hardware and loading the OS image, context switch code, configuration and access to hardware resources. The HAL is commonly referred to by platform vendors as drivers or BSPs. Most companies already provide such libraries to allow use of specialized hardware. This layer enables the STRS infrastructure to have a direct interface to the hardware drivers on the platform.

There are two requirements concerning the HAL in the STRS architecture:

- a. STRS-11 requires a HAL software API, which defines the physical and logical interfaces for intermodule and intramodule integration. The HAL is required for communicating data and control information between the GPP and the specialized hardware. The HAL API is not currently defined in this Standard but is left for the STRS platform provider to specify.*
- b. STRS-92 requires HAL documentation that includes a description of each method, its calling sequence, the return values, an explanation of the functionality, preconditions for using the method, postconditions after using the method, and examples where helpful. Note that the delivery of the HAL source code is not required.*

NASA-STD-4009

The electrical interfaces, connector requirements, and physical requirements are specified by the STRS platform provider in the HID. Information on a module's use of data in the HID will be made available to STRS application developers, either directly from the manufacturer (for specific types of components) or from the STRS platform provider (for memory maps based on electrical connections). The infrastructure or HAL may use this information to appropriately initialize hardware drivers such that control and data messages are delivered to the module.

Even though there is not a requirement for the STRS OE to be portable, the HAL is expected to foster portability and reusability of the STRS infrastructure and specialized hardware in different combinations from that originally designed. It can reduce the design efforts otherwise necessary to adapt the software to a new hardware platform. The goal with the HAL is to make it easier to change or add new hardware and to minimize the impact to the software. It does this by localizing the differences in software so that most of the STRS OE code does not need to be changed to run on a new platform or a platform with a new module.

Table 60, Sample HAL Documentation, shows an example of the HAL API for the function OPEN.

(STRS-92) The STRS platform provider shall provide the STRS platform HAL documentation that includes the following:

- (1) For each method or function, its calling sequence, return values, an explanation of its functionality, any preconditions for using the method or function, and the postconditions after using the method or function.
- (2) Information required to address the underlying hardware, including the interrupt input and output, the memory mapping, and the configuration data necessary to operate in the STRS platform environment.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Table 60—Sample HAL Documentation

HAL API	RESULT OPEN(HANDLE* resourceHandle, RESOURCE_NAME resourceName)
Description	Open a resource by name. If no errors are encountered, use the resourceHandle to access the resource.
Parameters	<ul style="list-style-type: none"> resourceHandle - [out] A pointer to place the opened handle into resourceName - [in] The name of the resource to open
Return	<p>A 32-bit signed integer used to determine whether an error has occurred. Use TEST_ERROR to obtain a printable message.</p> <ul style="list-style-type: none"> Zero - No errors or warnings. Positive – Warning. Negative – Error.
Precondition	Resource is not open before executing this command.
Postcondition	Resource will be open and ready for further access if no error was encountered.
See Also	READ, WRITE, CLOSE, TEST_ERROR
Example	<pre>#include <HALResources.h> ... RESULT result; HANDLE resourceHandle; RESOURCE_NAME resourceName = "FPGA"; result = OPEN(&resourceHandle, resourceName) if (result < 0) { cout << "Error: " << TEST_ERROR(result) << endl; } else if (result > 0) { cout << "Warning: " << TEST_ERROR(result) << endl; }</pre>

8. EXTERNAL COMMAND AND TELEMETRY INTERFACES

An STRS radio cannot perform the necessary application and platform functions without an external system providing commands, accepting responses, and monitoring the radio's health and status. The STRS radio implements an external interface to receive and act on the commands from the external system, translates the commands into the format expected by the application, and provides the information for monitoring the health and status of the radio. If the STRS radio has the capability for new or modified OE, application software, or configurable hardware design, the external command and telemetry interfaces should be able to accept and store new files. The interface in the STRS radio and in the external system, which is to provide the control, via a command sequence, to the STRS radio and receive responses from an STRS radio, is referred to as the STRS command and telemetry interfaces. The external STRS command and telemetry functionality illustrated in figure 17, Command and Telemetry Interfaces, typically resides on the spacecraft's flight computer, and/or it may reside on a ground station or another spacecraft.

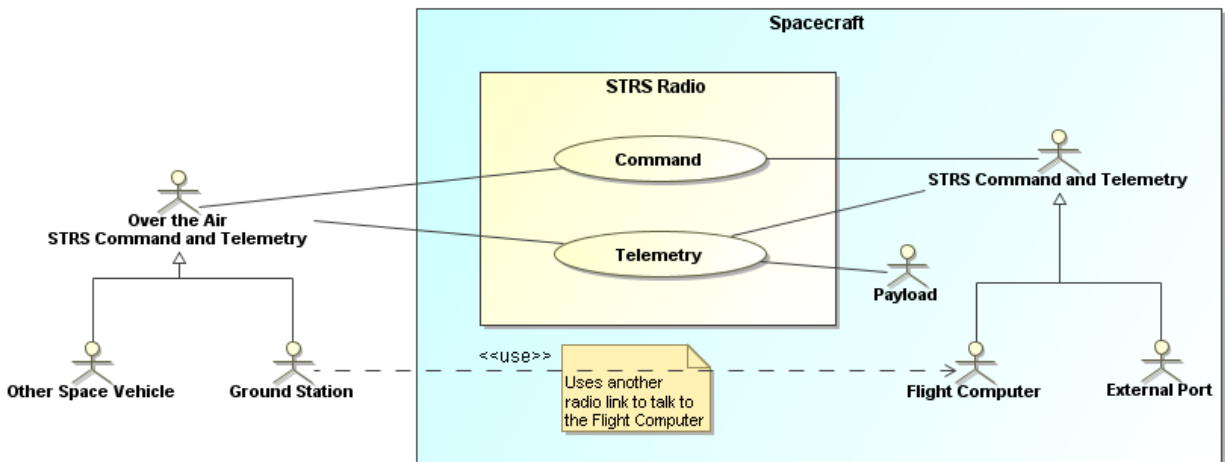


Figure 17—Command and Telemetry Interfaces

This shared capability implies that the STRS radio is capable of performing the interface functions. Within the STRS radio, if there are data stored on the radio that are to be transferred to an external system, the capability is to exist to send data using a mission-specific protocol to the receiver (flight computer, ground station, or other spacecraft) and capability in the receiver to process those data or write those data to a file or download service or to a storage area that is accessible from both. The reverse capability for STRS radio control is also necessary: The external system is capable of sending commands using a mission-specific protocol and the STRS radio is capable of validating, deciphering, and processing those commands. For example, data coming over the Flight Computer Interface are interpreted by the Command and Control Manager as shown in figure 13 and are processed by the STRS infrastructure.

Within the STRS radio, components of the command and telemetry interfaces are necessary to provide the interfaces between the STRS OE and the STRS command and telemetry functionality

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

on the external system. The command and telemetry interfaces may include a standard type of mechanical, electrical, and functional spacecraft bus interface, such as MIL-STD-1553; command and telemetry interpretation; and translation of the command set to the STRS standard necessary for application control. The protocol, command set, and telemetry set for the STRS command and telemetry interfaces are NOT part of the STRS standard but can be unique to each mission. A number of interface and behavior requirements are part of the standard to support the mission-specific protocols.

The requirements related to the external command and telemetry interfaces follow:

(STRS-94) An STRS platform shall accept, validate, and respond to external commands.

(STRS-95) An STRS platform shall execute external application control commands using the standardized STRS APIs.

(STRS-107) An STRS platform provider shall document the external commands describing their format, function, and any STRS methods invoked.

If an STRS application needs to interface with an external system request or provide telemetry, the following requirements apply:

(STRS-96) The STRS infrastructure shall use the STRS_Query method to service external system requests for information from an STRS application.

The STRS telemetry set will be mission-specific but will likely contain some or all of the following parameters:

- a. Power values.*
 - (1) Voltage, current, and power readings.*
- b. Environment values.*
 - (1) Temperature.*
 - (2) Pressure.*
- c. Power on reset test result status.*
 - (1) RAM test.*
 - (2) Read-only memory (ROM) test.*
 - (3) File management test.*
 - (4) PROM software revision.*
 - (5) Maximum memory configuration.*
 - (6) Individual module self-test status (GO/NO GO).*
- d. Module configuration.*
 - (1) Module type.*
 - (2) Module location.*
 - (3) Hardware revision.*

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

- e. *Application-specific parameters.*
- f. *Language support (C and/or C++).*
- g. *STRS Architecture Standard version.*
- h. *STRS OE release version.*
- i. *Available memory and free space for data and files.*

A suggested set of services that may be implemented by the STRS command and telemetry interfaces on the external system (flight computer, ground station, or other spacecraft) is shown in table 61, Suggested Services Implemented by the STRS command and telemetry interfaces. These services are NOT required for the STRS Architecture Standard at this time, but are likely needed for commanding and controlling an SDR and are expected to be part of the external system set of required functions.

**Table 61—Suggested Services Implemented by the
STRS Command and Telemetry Interfaces**

Function	Description
Application Control	
Application Selection	This command requests that the STRS radio instantiate the application and facilitate the installation of devices and resources requested by the application. This service should not impact existing applications. The command arguments will include the application ASCII name of a deployed configuration file that identifies all other files and initial parameters specified for an application.
Application Configuration	This command requests a customization of the application by specifying parameters the application will use.
Application Query	This command requests the current parameters and operational values of the application.
Application Start	This command requests that an initialized application begin processing application data. If the application has not been selected or completed initialization, the command will be rejected.
Application Stop	This command requests that a running application halt processing of application data. The application resources are not deallocated.
Application Unload	This command requests that the STRS infrastructure unload the identified application and release all resources associated with the application.
File Control Interface	
Upload File Request	This request will initiate an upload of a file to the STRS radio and place it in a specified location. If the command gets an error, the reason will be made available.
Delete File Request	This is a request for the deletion of a specified file from an STRS platform.
Download File Request	This request is complementary to the Upload File Request. This command will initiate a download of a specified file from the STRS platform.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Radio Control Interface	
Built-in-test	This request will perform a commanded built-in-test used to monitor the health of the radio and diagnose any problems.
Telemetry Control Interface	
Telemetry Control	Several different telemetry structure definitions may exist for different classes of STRS radios. Many systems will employ a polling technique where the data are provided only upon request. Other systems may desire a grouping of telemetry that can be identified to be sent at some periodic rate.

9. CONFIGURATION FILE(S)

Configuration files are used by the STRS infrastructure to specify attributes of files, devices, queues, waveforms, and services contained on an STRS radio. Two types of configuration files are discussed, as follows: (1) Platform configuration files (which are optional); and (2) Application configuration files (which are required). Platform configuration files provide the STRS infrastructure with information on the devices and modules currently installed in the system. Application configuration files contain application-specific information for configuration and customization of installed applications, as well as information for the STRS infrastructure to use to instantiate applications on the radio GPP. Application configuration files provide STRS application developers with flexibility in choosing parameters and values deemed pertinent to the implementation unrestricted by the STRS platform providers.

9.1 General Configuration File Format Definition and Use

The use of XML version 1.0 to define the STRS platform and application configuration data allows STRS platform providers and STRS application developers to take advantage of the features of XML; that is, to have the ability to identify configuration information in a standard, human-legible, precise, flexible, and adaptable method. XML is a markup language for documents containing structured information that contains both content and some indication of what role that content plays. XML defines tags containing or delimiting content and showing the relationships between them (see <http://www.w3.org/XML/>). XML is used to hold data and metadata and is currently being used throughout the Joint Tactical Radio System (JTRS)–SCA development environment process. The XML-formatted version of the STRS platform and application configuration files is not intended to be sent directly to the radio because of the extra overhead required to transmit and process XML-formatted data. Instead, it is anticipated that the XML configuration file will be preparsed, and additional error checking on the file will be performed prior to transmission. This process will reformat the configuration file into an appropriately optimized configuration file, which will subsequently be loaded into the radio. Requirements and discussion related to the configuration files refer to both the predeployed (i.e., nonoptimized XML file) configuration files and deployed (i.e., optimized) configuration files. The platform developers have the option of specifying the predeployed files as the deployed configuration files. For consistency and simplicity, XML 1.0 is required. The use of XML 1.0 for the application configuration files is required; it is strongly encouraged for the development of the platform configuration files.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

There are at least two options for preprocessing the XML domain profile for the STRS architecture:

- (1) Generate actual code by the preprocessor to deploy the application onto the specific hardware.*
- (2) Convert the XML domain profile into a static binary format that would be input to an application deployment routine that loads the application.*

The first option has the benefit of deploying the application as fast as possible, since the deployment code is specific to the application on the specific platform. The disadvantage of this approach would be that the deployment code would have to be regenerated for all applications that move to a different platform. The second option provides a more flexible approach, such that the XML files are translated into a standard binary format used by all applications and platforms. If the platform changes for a group of applications, only a new deployment routine has to be created for the new platform and nothing has to be generated for each specific application.

The XML format can accommodate a number of required configuration parameter features, such as the following:

- a. Range limits of configuration parameters.*
- b. Discrete allowable values of data items.*
- c. Output formatting for each parameter that is specific to a mission.*
- d. Configuration parameter dependency logic.*
- e. Error-checking logic.*

An XML interface tool could be used to create and modify platform and application configuration files. Commercially available XML interface tools provide an interface for basic editing of the configuration data files. In addition, these tools enforce error checking and interdependency checks to ensure that the entered data are correct and within the hardware and software limits. An XML Schema Definition (XSD) file contains an XML schema describing the structure and constraining the content of XML documents (See <http://www.w3.org/XML/Schema>). An XML schema is to be used to describe the XML file format of the application configuration files. Many tools use an XML schema to standardize the XML data entry and provide basic error checking.

Figure 18, XML Transformation and Validation, illustrates the relationships between an XML file and its corresponding schema, as well as representing the preprocessing of the XML file in a simplified form using Extensible Stylesheet Language (XSL) Transformations (XSLTs). XSL is a family of recommendations for defining XML document transformation and presentation. XSLT is a language for transforming XML into text using any other vocabulary imaginable. The XSLT uses an expression language, XML Path Language (XPath), to access or refer to parts of an XML document. For transmuting instances of configuration files in XML, to create the desired output, an XSL (XSLT and XPath) could be used (see <http://www.w3.org/Style/XSL/>).

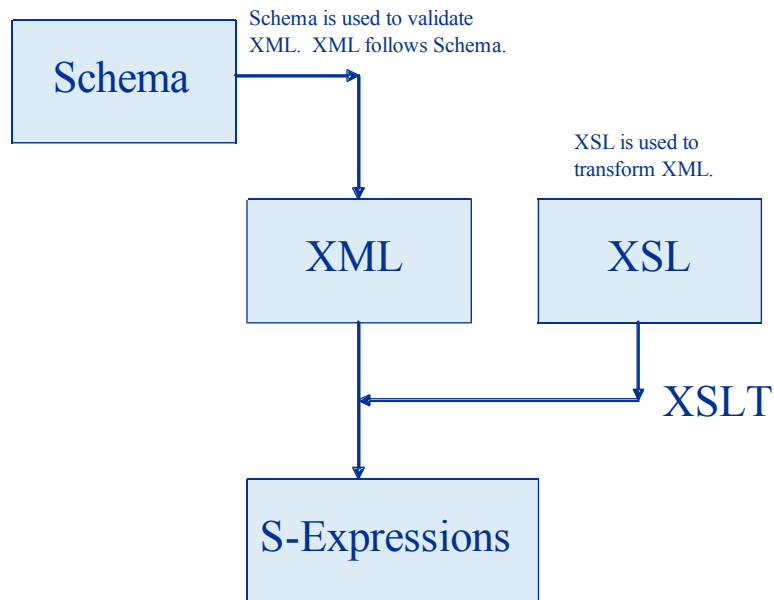


Figure 18—XML Transformation and Validation

The XML should be preprocessed to a platform-specific format to optimize space on the STRS radio while keeping the equivalent content.

The application configuration files are developed by the STRS integrators using information obtained from both the STRS platform provider and the STRS application developers. The STRS integrators use the application configuration files to install the applications on the platform. There may be multiple STRS integrators. The STRS integrator for each application may be the STRS platform provider or STRS application developer or a designated entity. The STRS integrator is always the STRS infrastructure developer for any applications delivered with the infrastructure. The application configuration file requirements are written assuming that the STRS application developers and STRS platform providers are separate entities and that not all the applications and documentation are available at the same time as the platform, schema, and transformation tools. Figure 19, Configuration File Development Process, details the process, provider, and related requirement numbers for the development and delivery of platform and application configuration files.

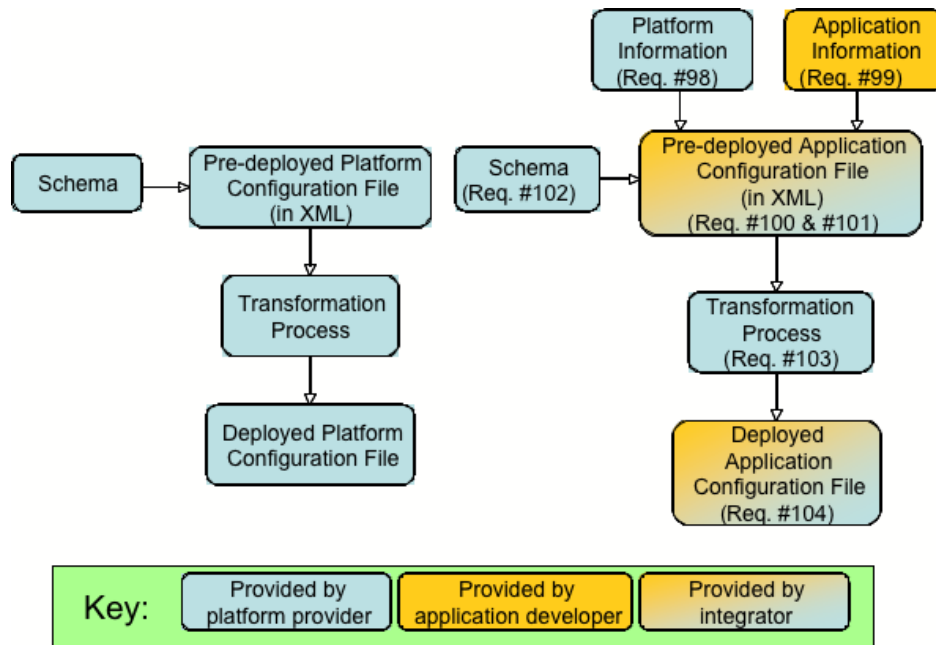


Figure 19—Configuration File Development Process

9.2 Platform Configuration Files

The development and delivery of the platform configuration files is a goal of the STRS architecture but is optional. The STRS platform provider has the option to choose the method to describe and use the hardware and software environment for the STRS infrastructure. Developing platform configuration file(s) is the likely method to be used by an STRS platform provider to identify the existence of the different hardware modules and their associated configuration files to allow the OE to instantiate drivers and test applications. An STRS platform configuration file may be used when starting the STRS infrastructure to configure various properties of the STRS platform. Configuring these properties at run-time allows greater flexibility than configuring them at compile-time. To increase the runtime flexibility of the STRS platform, the STRS infrastructure is likely to use deployed platform configuration files to determine the existence and attributes of the files, devices, queues, waveforms, and services contained on the STRS radio. Attributes of files, devices, and queues could include access (read/write, both, or append), type (text or binary), and other properties. The name of the starting configuration file(s) may be provided to the STRS infrastructure upon initialization. The predeployed platform configuration files should contain platform configuration information such as the following:

- a. Hardware module names and types.
- b. Memory types, sizes, and access.
- c. Memory mapping.
- d. Unique names and attributes of files, devices, queues, services, and applications known to the OE at boot-up.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

An XML 1.0 schema should be provided with the predeployed platform configuration files to validate the format and data in the XML configuration file. The XML schema is usually a separate file so that multiple configuration files can reference the same schema. The XML schema for the platform should contain information to validate the order of the tags, the number of occurrences of each tag, and the values or attributes. The XML schema for the platform configuration files should ensure the following:

- a. Numeric values are valid numbers and fall within an allowable range.*
- b. Alphabetic values are strings of characters and, if appropriate, are chosen from a given set.*
- c. Hexadecimal values conform to rules for hexadecimal numbers using “digits” from the set {0123456789abcdef} and fall within an allowable range.*

To support the need to upgrade or modify the platform, the STRS platform provider should provide the following platform configuration file artifacts with the platform:

- a. Predeployed platform configuration file.*
- b. XML schema to validate the format and data in the corresponding predeployed STRS platform configuration files, including the order of the tags, the number of occurrences of each tag, and the values or attributes.*
- c. Tools and documentation for the transformation of a predeployed platform configuration file in XML into a deployed platform configuration file.*
- d. Deployed STRS platform configuration file.*

9.3 Application Configuration Files

A predeployed STRS application configuration file is created by the STRS integrator using platform information, the XML schema supplied by the STRS platform provider, and application information provided by the STRS application developer. The deployed application configuration file is used by the infrastructure (see the STRS_InstantiateApp method) when starting the STRS application to configure various properties of the STRS application. Configuring these properties at run time allows greater flexibility than configuring them at compile time. For example, one might configure the STRS handle names of files, devices, queues, waveforms and services needed by the STRS application so that these can be easily changed. Since a service is actually an application that has been incorporated into the STRS infrastructure, the format of the application configuration file should be a subset of the format of the platform configuration file as specified by the schema. If any STRS application resources need to be loaded separately into memory or into a device, such as an FPGA, before the STRS application can function properly, these should be specified in the configuration file for that STRS application.

NASA-STD-4009

A predeployed STRS application configuration file is to be written in XML 1.0 to describe and save application configuration information. An XML schema is to be provided with the predeployed STRS application configuration files to validate their format and data. The XML schema is usually a separate file so that multiple STRS application configuration files can reference the same schema. The XML schema for the STRS application should contain information to validate the order of the tags, the number of occurrences of each tag, and the values or attributes. The XML schema for the STRS application configuration files should also ensure the following:

- a. Numeric values are valid numbers and fall within an allowable range.*
- b. Alphabetic values are strings of characters and, if appropriate, are chosen from a given set.*
- c. Hexadecimal values conform to rules for hexadecimal numbers using “digits” from the set {0123456789abcdef} and fall within an allowable range.*

The operational parameters specified are used during the operation of the radio to initialize or reinitialize the STRS application into a known state using the STRS_Configure and APP_Configure methods. The STRS application should be automatically restarted into this known state after any problem that requires cycling power.

(STRS-98) The STRS platform provider shall document the necessary platform information (including a sample file) to develop a predeployed application configuration file in XML 1.0.

(STRS-99) The STRS application developer shall document the necessary application information to develop a predeployed application configuration file in XML 1.0.

(STRS-100) The STRS integrator shall provide a predeployed application configuration file in XML 1.0.

(STRS-101) The predeployed STRS application configuration file shall identify the following application attributes and default values:

- (1) Identification.
 - A. Unique STRS handle name for the application.
 - B. Class name (if applicable).
- (2) State after processing the configuration file.
- (3) Any resources to be loaded separately.
 - A. Filename of loadable image.
 - B. Target on which to put loadable image file.
 - C. Target memory in bytes, number of gates, or logic elements.
- (4) Initial or default values for all distinct operationally configurable parameters.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

(STRS-102) The STRS platform provider shall provide an XSD file to validate the format and data for predeployed STRS application configuration files, including the order of the tags, the number of occurrences of each tag, and the values or attributes.

(STRS-103) The STRS platform provider shall document the transformation (if any) from a predeployed application configuration file in XML into a deployed application configuration file and provide the tools to perform such transformation.

(STRS-104) The STRS integrator shall provide the deployed STRS application configuration file for the STRS infrastructure to place the STRS application in the specified state.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

APPENDIX A

EXAMPLE CONFIGURATION FILES

A.1 STRS Platform Configuration File Hardware Example

Appendix A introduces examples of platform and application configuration files, necessary for application execution and platform initialization. Appendix A also describes example configuration file formats. STRS configuration files contain platform- and application-specific information for the customization of installed applications. These examples are not required formats. They are intended to illustrate some considerations that STRS platform providers and STRS application developers should take into account when designing their configuration file formats.

An example of the format of the portion of an STRS platform configuration file that deals with hardware is implemented in an XML schema. This format is shown in figure 20, Example of Hardware Portion of STRS Platform Configuration File.

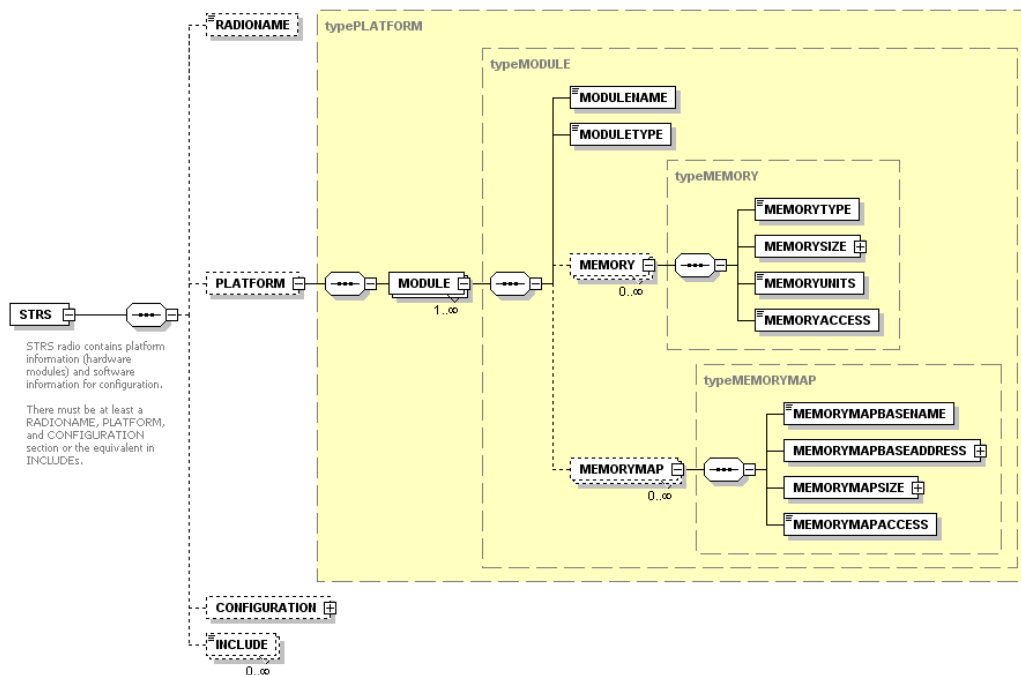


Figure 20—Example of Hardware Portion of STRS Platform Configuration File

For any GPP, the memory size and memory location should be specified in bytes. *Rationale for International Standard—Programming Language—C states the following:*

- (1) “All objects in C must be representable as a contiguous sequence of bytes, each of which is at least 8 bits wide.”

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

- (2) “Any object can be treated as an array of characters, the size of which is given by the sizeof operator with that object’s type as its operand.”
- (3) “It is fundamental to the correct usage of functions, such as malloc and fread that sizeof(char) be exactly one.”

Therefore, for consistency across C and C++ implementations, bytes are used.

MODULE list	A list of hardware modules having memory able to contain data and executable software.
-------------	--

• MODULENAME	The unique name for each hardware module accessible from the current GPP. The current GPP is denoted by SELF.
• MODULETYPE	The name of the hardware type. The hardware module types may be GPP, RF, FPGA, DSP, ASIC, and so forth.

• MEMORY list	A list of memory areas of various types. See below for further information.
○ MEMORYTYPE	Memory type may be RAM, EEPROM, etc.
○ MEMORYSIZE	The number of memory units.
○ MEMORYUNITS	Memory units may be BYTES or, GATES. For any GPP, the size is to be in BYTES.
○ MEMORYACCESS	Memory access for the memory. Access may be READ, WRITE, or BOTH.

• MEMORYMAP list	This list provides the base addresses and memory size of regions of the current GPP RAM (SELF) that are memory mapped to the module: that is, memory mapped to an external device. There may be more than one item in the list when different parts of memory are either not contiguous or are used for different purposes. See section A.2, under DEVICE list, in ATTRIBUTE list, for memory offsets specific to the device associated with a name.
○ MEMORYBASENAME	A unique identifier for the portion of memory mapped to the module.
○ MEMORYBASEADDRESS	The starting byte address reserved for memory mapping.
○ MEMORYSIZE	Number of bytes starting at the base address reserved for memory mapping.
○ MEMORYACCESS	Memory access for the portion of memory mapped to the module. Access may be READ, WRITE, or BOTH. The access defined here may be different from the memory access defined in the previous section when part of the memory is used for one purpose and another part is used for a different purpose.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

A.2 STRS Platform Configuration File Software Example

An example of the format of the portion of an STRS platform configuration file that deals with software is implemented in an XML schema. This format is shown in figure 21, Example of Software Portion of STRS Platform Configuration File.

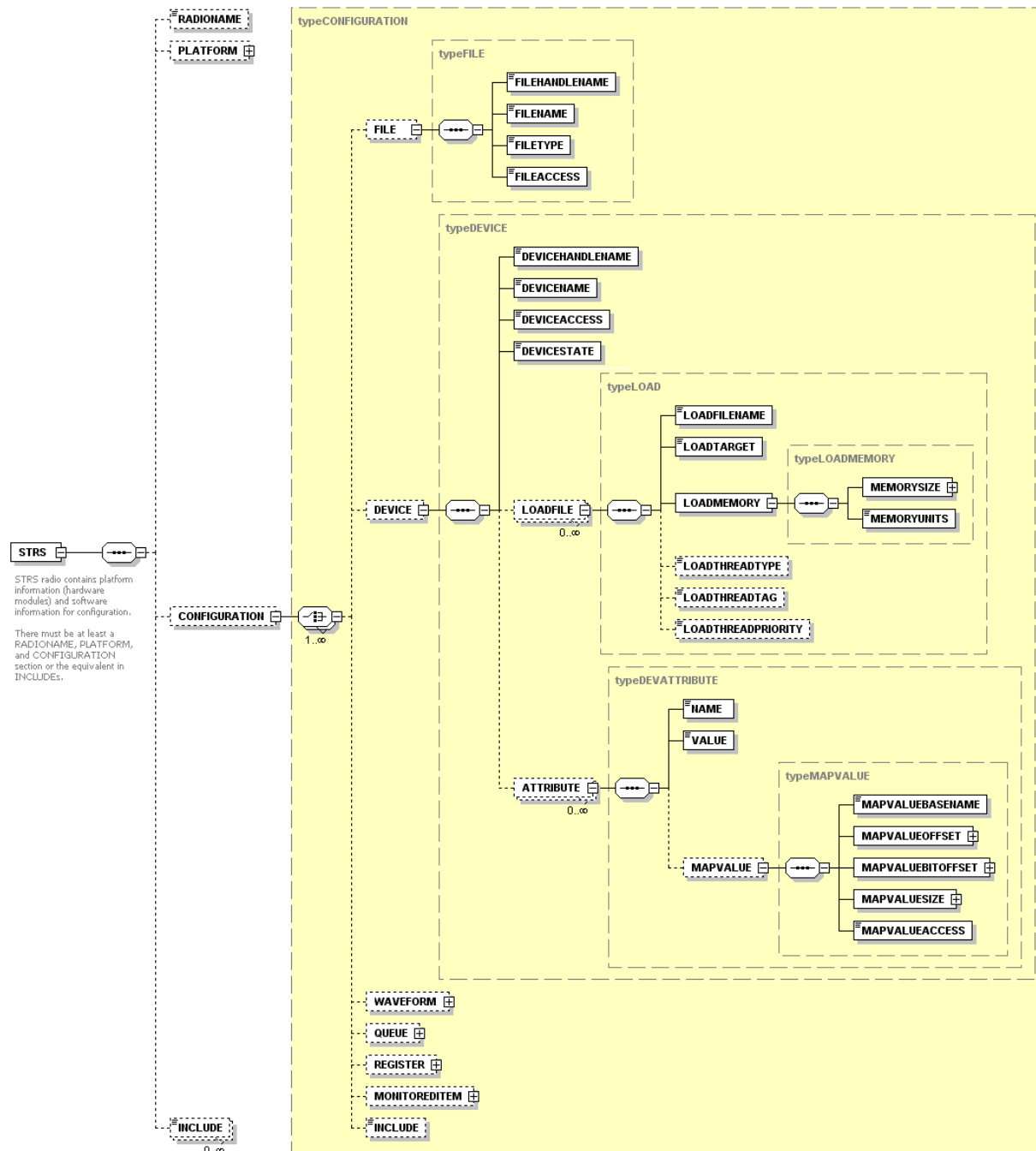


Figure 21—Example of Software Portion of STRS Platform Configuration File

NASA-STD-4009

FILE list	A list of files to read, write, both, or append from multiple locations using a handle ID.
• FILEHANDLENAME	This is usually a unique shortened form of the file name used in messages and for obtaining the handle ID.
• FILENAME	Storage area name or fully qualified file name.
• FILETYPE	The file type may be TEXT or BINARY.
• FILEACCESS	The file access may be READ, WRITE, BOTH, or APPEND. BOTH implies update: that is, READ and WRITE.
DEVICE list	A list of devices to read or write from multiple locations using a handle ID. A device in the list is software that acts as a proxy for some hardware connection to an external device or a software manager for access to multiple or variable devices.
• DEVICEHANDLENAME	This is usually a unique shortened form of the module name used in messages and for obtaining the handle ID.
• DEVICENAME	This is usually a shortened form of the module name for the device. If coded in C++, this is the class name.
• DEVICEACCESS	The access to the device may be specified as READ, WRITE, BOTH, or NONE. READ indicates that the device implements APP_Read(). WRITE indicates that the device implements APP_Write().
• LOADFILE list	A list of files to be loaded for execution if not already loaded. Usually, the software for the device on the current GPP (SELF) should be loaded before the configurable hardware design so that the software can load and configure the device as necessary.
○ LOADFILENAME	Storage area name or fully qualified file name.
○ LOADTARGET	The module name for the device on which the file is instantiated or loaded. The load process is determined by the corresponding MODULE information (see section A.1).
○ LOADMEMORY	
▪ MEMORYSIZE	The number of memory units.
▪ MEMORYUNITS	Memory units may be BYTES or GATES. For any GPP, the size is to be in BYTES.
○ LOADTHREADTYPE	
○ LOADTHREADTAG	
○ LOADTHREADPRIORITY	
• ATTRIBUTE list	A list of properties set as default during initialization.
○ NAME	Name of the attribute.
○ VALUE	Value of the attribute.
○ MAPVALUE list	Location in memory of the attribute when memory mapped. A location is to be unique to the associated device.
▪ MAPVALUEBASENAME	A unique identifier for the portion of memory mapped to the module. This is to match a MEMORYBASENAME value defined in section A.1, under MODULE list in the MEMORYMAP list.
▪ MAPVALUEOFFSET	Offset from the address of baseName as defined in the module list's memory map list.
▪ MAPVALUEBITOFFSET	Bit offset from the high order position to begin.
▪ MAPVALUESIZE	Number of bits in which to store the value.
▪ MAPVALUEACCESS	Memory access may be READ, WRITE, or BOTH.
QUEUE list	The information necessary to create queues.
• QUEUEHANDLENAME	The name of the queue that the publisher uses to send data to the subscribers. Used in messages and for obtaining the handle ID.
• QUEUEATYPE	READ for pull, WRITE for push. In all cases, STRS_Write is used to write to the queue. READ indicates that STRS_Read is used to

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

	obtain data from the queue. WRITE indicates that the queue calls STRS_Write to send the data to any subscribers.
• QUEUEPRIORITY	Priority of queue.
REGISTER list	The correspondences between queues and subscribers. This decouples publishers from subscribers.
• PUBLISHER	The name of the queue that the publisher uses to send data to the subscribers. Used in messages and for obtaining the handle ID.
• SUBSCRIBER	A handle name for a subscriber. Used in messages and for obtaining the handle ID.
MONITOREDITEM list	A list of monitored items that are tested to indicate the health of the system.
• ATTRIBUTENAME	The name of the property whose value is to be tested in a monitored component.
• HANDLENAME	The handle name defines the monitored component from which to obtain the value corresponding to the attributeName.
• DELAY	A positive value represents the nominal time delay between successive automated tests of the monitored component. A nonpositive value indicates that the test is to be requested.
• TESTTYPE	The type of test to apply to the property to ascertain whether the value indicates the monitored component is healthy. Examples include testing for exact values, within ranges, or by use of operations in Reverse Polish Notation (RPN).
○ EXACT	Monitored value is to be one of the values in the value list.
○ EXCLUDE	Monitored value is not to be in the value list.
○ BETWEENII	Monitored value is to be between the pairs of values in the value list including both end points.
○ BETWEENIX	Monitored value is to be between the pairs of values in the value list including the low end point and excluding the high end point.
○ BETWEENXI	Monitored value is to be between the pairs of values in the value list excluding the low end point and including the high end point.
○ BETWEENXX	Monitored value is to be between the pairs of values in the value list excluding both end points.
○ RPN	The attributeName, values to be tested, and operators is to appear in the value list using RPN. RPN uses sequences of one or two arguments followed by an operator. The result of applying the operator replaces the original sequence used, and the process is repeated until there are no more operators. The attributeName for the monitored value is replaced, in the RPN formula, by the corresponding property value. For example, the sequence of data and operators in the VALUE list for testing the property named D in RPN—0;D;LT;D;500;LE;AND—is equivalent to (0<D && D<500)
	The current set of operators includes the following: AND, OR, XOR, NOT, EQ, NE, GT, GE, LT, LE, PLUS, MINUS, MULTIPLY, DIVIDE, MOD, MIN, MAX, If floating point is required or allowed, the set of operators could be augmented with the following: SIN, COS, TAN, ASIN, ACOS, ATAN1, ATAN2, SINH, COSH, TANH, ABS, EXP, LOG10, LN, SQRT, FLOOR, CEIL, ROUND, POW.
• VALUE list	A list of values and possibly operations used corresponding to the value of TESTTYPE. <ul style="list-style-type: none"> ○ For example, if TESTTYPE is EXACT, the VALUE list will contain {512,1024,2048,4096} if those are the allowed values. ○ If TESTTYPE is EXCLUDE and odd numbers between 1 and 10 are not allowed, the VALUE list will contain {1,3,5,7,9}.

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

-
- If TESTTYPE is BETWEENII and the attribute D is allowed between 0 and 500, inclusive ($0 < D < 500$), the VALUE list will contain {0,500}. Because the TESTTYPE is BETWEENII, the lower limit, 0, is included and the upper limit, 500, is included.
 - If TESTTYPE is BETWEENIX and the attribute D is allowed between 0 and 500 ($0 < D < 500$), the VALUE list will contain {0,500}. Because the TESTTYPE is BETWEENIX, the lower limit, 0, is included and the upper limit, 500, is excluded.
 - If TESTTYPE is BETWEENXI and the attribute D is allowed between 0 and 500 ($0 < D < 500$), the VALUE list will contain {0,500}. Because the TESTTYPE is BETWEENXI, the lower limit, 0, is excluded and the upper limit, 500, is included.
 - If TESTTYPE is BETWEENXX and the attribute D is allowed between 0 and 500, exclusive ($0 < D < 500$), the VALUE list will contain {0,500}. Because the TESTTYPE is BETWEENXX, the lower limit, 0, is excluded and the upper limit, 500, is excluded.
 - If TESTTYPE is RPN and the attribute D is allowed between 0 and 500 ($0 < D < 500$), the VALUE list will contain {0,D,LT,D,500,LE,AND}.
-

A.3 STRS Application Configuration File Example

An example of the format of an STRS application configuration file in XML is shown in Figure 22, Example of STRS Waveform Configuration File.

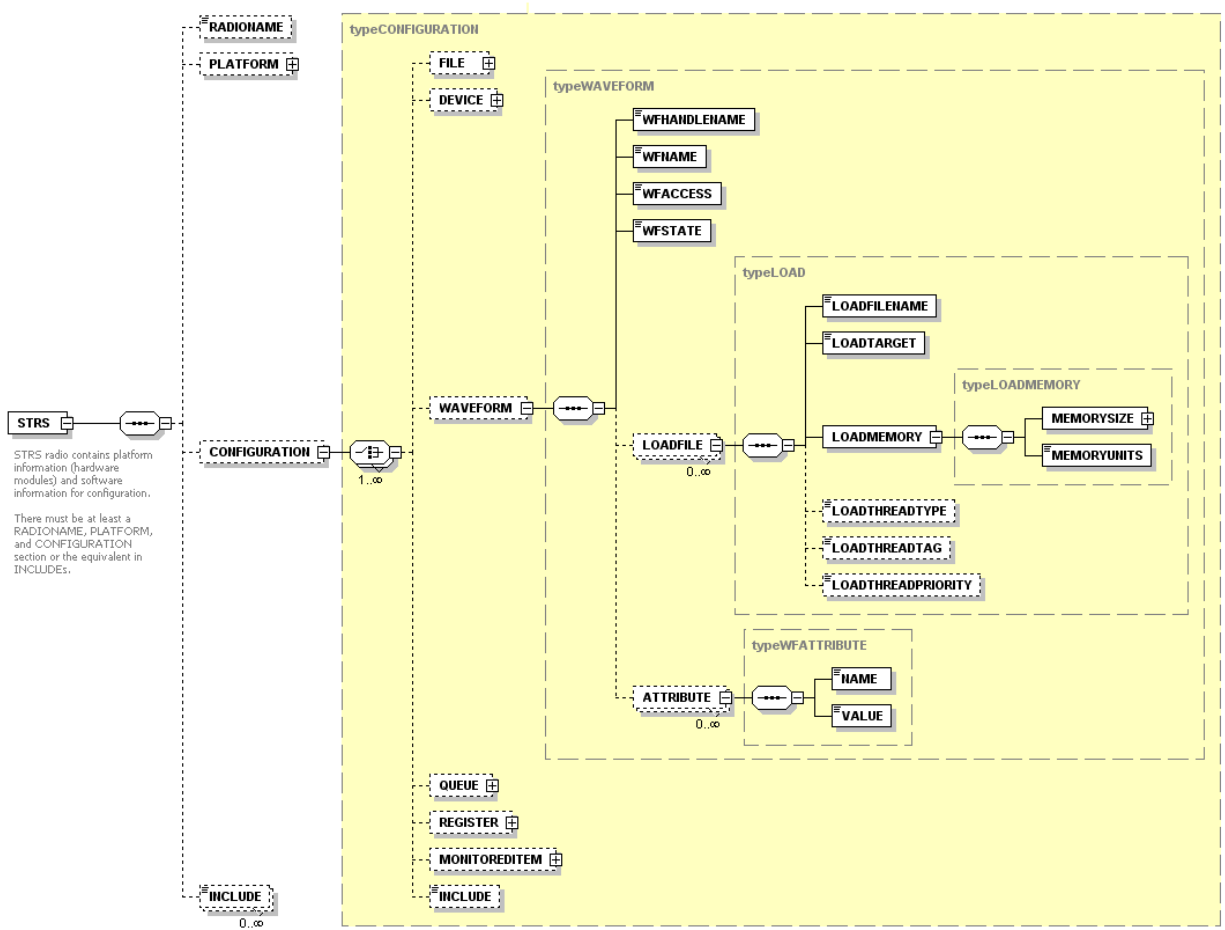


Figure 22—Example of STRS Waveform Configuration File

NASA-STD-4009

APPLICATION list	
• WFHANDLENAME	This is a unique shortened form of the application name used in messages and for obtaining the handle ID.
• WFNAME	If coded in C++, this is the application class name.
• WFACCESS	The access to the application may be specified as READ, WRITE, BOTH, or NONE. READ indicates that the application implements APP_Read(). WRITE indicates that the application implements APP_Write().
• WFSTATE	The state at which the application is left after processing the configuration file. The state may be STRS_APP_INSTANTIATED, STRS_APP_STOPPED, or STRS_APP_RUNNING.
• LOADFILE list	A list of files to be loaded for execution if not already loaded. Usually, the software for the application on the current GPP (SELF) should be loaded before the configurable hardware design so that the software can load and configure the software or configurable hardware design as necessary.
○ LOADFILENAME	Storage area name or fully qualified file name
○ LOADTARGET	Module name for the device on which the file is instantiated. The load process is determined by the corresponding MODULE information (see A.1).
○ LOADMEMORY	
• MEMORYSIZE	The number of memory units.
• MEMORYUNITS	Memory units may be BYTES or, GATES. For any GPP, the size is to be in BYTES.
○ LOADTHREADTYPE	
○ LOADTHREADTAG	
○ LOADTHREADPRIORITY	
• ATTRIBUTE list	A list of properties set as default during initialization.
○ NAME	Name of the attribute
○ VALUE	Value of the attribute

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

APPLICATION list	
• WFHANDLENAME	This is a unique shortened form of the application name used in messages and for obtaining the handle ID.
• WFNAME	If coded in C++, this is the application class name.
• WFACCESS	The access to the application may be specified as READ, WRITE, BOTH, or NONE. READ indicates that the application implements APP_Read(). WRITE indicates that the application implements APP_Write().
• WFSTATE	The state at which the application is left after processing the configuration file. The state may be STRS_APP_INSTANTIATED, STRS_APP_STOPPED, or STRS_APP_RUNNING.
• LOADFILE list	A list of files to be loaded for execution if not already loaded. Usually, the software for the application on the current GPP (SELF) should be loaded before the configurable hardware design so that the software can load and configure the software or configurable hardware design as necessary.
○ LOADFILENAME	Storage area name or fully qualified file name
○ LOADTARGET	Module name for the device on which the file is instantiated. The load process is determined by the corresponding MODULE information (see A.1).
○ LOADMEMORY	
• MEMORYSIZE	The number of memory units.
• MEMORYUNITS	Memory units may be BYTES or, GATES. For any GPP, the size is to be in BYTES.
○ LOADTHREADTYPE	
○ LOADTHREADTAG	
○ LOADTHREADPRIORITY	
• ATTRIBUTE list	A list of properties set as default during initialization.
○ NAME	Name of the attribute
○ VALUE	Value of the attribute

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

APPENDIX B

POSIX API PROFILE

Appendix B provides a list of the POSIX profile recommended as part of the application abstraction.

Table 62, POSIX Subset Profiles PSE51, PSE52, and PSE53 provides the POSIX subset in profiles PSE51, PSE52, and PSE53.

Table 62—POSIX Subset Profiles PSE51, PSE52, and PSE53

Unit of Functionality	Interfaces	PSE51	PSE52	PSE53
POSIX_C_LANG_JUMP	longjmp(), setjmp()	X	X	X
POSIX_C_LANG_MATH	acos(), acosf(), acosh(), acoshf(), acoshl(), acosl(), asin(), asinf(), asinh(), asinhf(), asinhl(), asinl(), catan(), atan2(), atan2f(), atan2l(), atanf(), atanh(), atanhf(), atanh(), atanl(), cabs(), cabsf(), cabsl(), cacos(), cacosf(), cacosh(), cacoshf(), cacoshl(), cacosl(), carg(), cargf(), cargl(), casin(), casinf(), casinh(), casinhf(), casinhl(), casinl(), catan(), catanf(), catanh(), catanhf(), catanh(), catanl(), cbrt(), cbrtf(), cbrtl(), ccos(), ccosf(), ccosh(), ccoshf(), ccoshl(),		X	X
POSIX_C_LANG_MATH	ccosl(), ceil(), ceilf(), ceil(), cexp(), cexpf(), cexpl(), cimag(), cimagf(), cimagl(), clog(), clogf(), clogl(), conj(), conjf(), conjl(), copysign(), copysignf(), copysignl(), cos(), cosf(), cosh(), coshf(), coshl(), cosl(), cpow(), cpowf(), cpowl(), cproj(), cprojf(), cprojl(), creal(), crealf(), creall(), csin(), csinf(), csinh(), csinhf(), csinhl(), csinl(), csqrt(), csqrtf(), csqrtl(), ctan(), ctanf(), ctanh(), ctanhf(), ctanh(), ctanl(), erf(), erfc(), erfcf(), erfcl(), erff(), erfl(), exp(), exp2(), exp2f(), exp2l(), expf(), expl(), expm1(), expm1f(), expm1l(), fabs(), fabsf(), fabsl(), fdim(), fdimf(), fdiml(), floor(), floorf(), floorl(), fma(), fmaf(), fmal(),		X	X

NASA-STD-4009

Unit of Functionality	Interfaces	PSE51	PSE52	PSE53
POSIX_C_LANG_MATH	fmax(), fmaxf(), fmaxl(), fmin(), fminf(), fminl(), fmod(), fmodf(), fmodl(), fpclassify(), frexp(), frexpf(), frexpl(), hypot(), hypotf(), hypotl(), ilogb(), ilogbf(), ilogbl(), isfinite(), isgreater(), isgreaterequal(), isinf(), isless(), islessequal(), islessgreater(), isnan(), isnormal(), isunordered(), ldexp(), ldexpf(), ldexpl(), lgamma(), lgammaf(), lgammal(), llrint(), llrintf(), llrintl(), llround(), llroundf(), llroundl(), log(), log10(), log10f(), log10l(), log1p(), log1pf(), log1pl(), log2(), log2f(), log2l(), logb(), logbf(), logbl(), logf(), logl(), lrint(), lrintf(), lrintl(), lround(), lroundf(), lroundl(), modf(), modff(), modfl(), nan(), nanf(), nanl(), nearbyint(), nearbyintf(), nearbyintl(), nextafter(), nextafterf(), nextafterl(), nexttoward(), nexttowardf(), nexttowardl(), pow(), powf(), powl(), remainder(), remainderf(), remainderl(), remquo(), remquoof(), remquoofl(), rint(), rintf(), rintl(), round(), roundf(), roundl(), scalbln(), scalblnf(), scalblnl(), scalbn(), scalbnf(), scalbnl(), signbit(), sin(), sinf(), sinh(), sinhlf(), sinhl(), sinl(), sqrt(), sqrtf(), sqrtl(), tan(), tanf(), tanh(), tanhf(), tanhl(), tanl(), tgamma(), tgammaf(), tgamma(), trunc(), truncf(), trunc()		X	X

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Unit of Functionality	Interfaces	PSE51	PSE52	PSE53
POSIX_C_LANG_SUPPORT	abs(), asctime(), asctime_r(), atof(), atoi(), atol(),atoll(), bsearch(), calloc(), ctime(), ctime_r(),difftime(), div(), feclearexcept(), fegetenv(), fegetexceptflag(), fegetround(), fholdexcept(),feraiseexcept(), fesetenv(), fesetexceptflag(), fesetround(), fetestexcept(), feupdateenv(), free(),gmtime(), gmtime_r(), imaxabs(), imaxdiv(), isalnum(), isalpha(), isblank(), iscntrl(), isdigit(),isgraph(), islower(), isprint(), ispunct(), isspace(),isupper(), isxdigit(), labs(), ldiv(), llabs(), lldiv(), localeconv(), localtime(), localtime_r(), malloc(),memchr(), memcmp(), memcpy(), memmove(),memset(), mktime(), qsort(), rand(), rand_r(), realloc(), setlocale(), snprintf(), sprintf(), srand(),sscanf(), strcat(), strchr(), strcmp(), strcoll(), strcpy(),strcspn(), strerror(), strerror_r(), strptime(), strlen(), strncat(), strncmp(), strncpy(), strpbrk(), strrchr(), strspn(), strstr(), strtod(), strtod(), strtoimax(),strtok(), strtok_r(), strtol(), strtold(), strtoll(), strtoul(), strtoull(), strtoumax(), strxfrm(), time(),tolower(), toupper(), tzname, tzset(), va_arg(),va_copy(), va_end(), va_start(), vsnprintf(), vsprintf(), vsscanf()	X	X	X
POSIX_DEVICE_IO	clearerr(), close(), fclose(), fdopen(), feof(), ferror(),fflush(), fgetc(), fgets(), fileno(), fopen(), fprintf(),fputc(), fputs(), fread(), freopen(), fscanf(), fwrite(),getc(), getchar(), gets(), open(), perror(), printf(),putc(), putchar(), puts(), read(), scanf(), setbuf(),setvbuf(), stderr, stdin, stdout, ungetc(), vfprintf(),vfscanf(), vprintf(), vscanf(), write()	X	X	X

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Unit of Functionality	Interfaces	PSE51	PSE52	PSE53
POSIX_EVENT_MGMT	FD_CLR(), FD_ISSET(), FD_SET(), FD_ZERO(), pselect(), select()			X
POSIX_FD_MGMT	dup(), dup2(), fcntl(), fgetpos(), fseek(), fseeko(), fsetpos(), ftell(), ftello(), ftruncate(), lseek(), rewind()		X	X
POSIX_FILE_LOCKING	flockfile(), ftrylockfile(), funlockfile(), getc_unlocked(), getchar_unlocked(), putc_unlocked(), putchar_unlocked()	X	X	X
POSIX_FILE_SYSTEM	access(), chdir(), closedir(), creat(), fpathconf(), fstat(), getcwd(), link(), mkdir(), opendir(), pathconf(), readdir(), readdir_r(), remove(), rename(), rewinddir(), rmdir(), stat(), tmpfile(), tmpnam(), unlink(), utime()		X	X
POSIX_MULTI_PROCESS	_Exit(), _exit(), assert(), atexit(), clock(), execl(), execle(), execlp(), execv(), execve(), execvp(), exit(), fork(), getpgrp(), getpid(), getppid(), setsid(), sleep(), times(), wait(), waitpid()			X

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Unit of Functionality	Interfaces	PSE51	PSE52	PSE53
POSIX_NETWORKING	accept(), bind(), connect(), endhostent(), endnetent(), endprotoent(), endservent(), freeaddrinfo(), gai_strerror(), getaddrinfo(), gethostbyaddr(), gethostbyname(), gethostent(), gethostname(), getnameinfo(), getnetbyaddr(), getnetbyname(), getnetent(), getpeername(), getprotobyname(), getprotobynumber(), getprotoent(), getservbyname(), getservbyport(), getservent(), getsockname(), getsockopt(), h_errno, htonl(), htons(), if_freenameindex(), if_indextoname(), if_nameindex(), if_nametoindex(), inet_addr(), inet_ntoa(), inet_ntop(), inet_pton(), listen(), ntohl(), ntohs(), recv(), recvfrom(), recvmsg(), send(), sendmsg(), sendto(), sethostent(), setnetent(), setprotoent(), setservent(), setsockopt(), shutdown(), socket(), socketatmark(), socketpair()			X
POSIX_PIPE	pipe()			X
POSIX_SIGNALS	abort(), alarm(), kill(), pause(), raise(), sigaction(), sigaddset(), sigdelset(), sigemptyset(), sigfillset(), sigismember(), signal(), sigpending(), sigprocmask(), sigsuspend(), sigwait()	X	X	X
POSIX_SIGNAL_JUMP	siglongjmp(), sigsetjmp()			X
POSIX_SINGLE_PROCESS	confstr(), environ, errno, getenv(), setenv(), sysconf(), uname(), unsetenv()	X	X	X

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Unit of Functionality	Interfaces	PSE51	PSE52	PSE53
POSIX_THREADS_BASE	pthread_atfork(), pthread_attr_destroy(), pthread_attr_getdetachstate(), pthread_attr_getschedparam(), pthread_attr_init(), pthread_attr_setdetachstate(), pthread_attr_setschedparam(), pthread_cancel(), pthread_cleanup_pop(), pthread_cleanup_push(), pthread_cond_broadcast(), pthread_cond_destroy(), pthread_cond_init(), pthread_cond_signal(), pthread_cond_timedwait(), pthread_cond_wait(), pthread_condattr_destroy(), pthread_condattr_init(), pthread_create(), pthread_detach(), pthread_equal(), pthread_exit(), pthread_getspecific(), pthread_join(), pthread_key_create(), pthread_key_delete(), pthread_kill(), pthread_mutex_destroy(), pthread_mutex_init(), pthread_mutex_lock(), pthread_mutex_trylock(), pthread_mutex_unlock(), pthread_mutexattr_destroy(), pthread_mutexattr_init(), pthread_once(), pthread_self(), pthread_setcancelstate(), pthread_setcanceltype(), pthread_setspecific(), pthread_sigmask(), pthread_testcancel()	X	X	X
POSIX_THREAD_MUTEX_EXT	pthread_mutexattr_gettype(), pthread_mutexattr_settype()	X	X	X
XSI_THREADS_EXT	pthread_attr_getguardsize(), pthread_attr_getstack(), pthread_attr_setguardsize(), pthread_attr_setstack(), pthread_getconcurrency(), pthread_setconcurrency()	X	X	X

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

APPENDIX C

REFERENCE DOCUMENTS

The following reference documents are recommended for further guidance.

Department of Defense

Document Number	Document Title
<u>SCA version 2 2 2.pdf</u>	Software Communications Architecture Specification

National Aeronautics and Space Administration

Document Number	Document Title
<u>NASA/TM—2007-215042</u>	Space Telecommunications Radio System (STRS) Architecture Goals/Objectives and Level 1 Requirements
<u>NASA/TP—2008-214813</u>	Space Telecommunications Radio System Software Architecture Concepts and Analysis
<u>NASA/TM—2008-215445</u>	Space Telecommunications Radio System (STRS) Definitions and Acronyms
<u>NASA/TM—2010-216809</u>	Space Telecommunications Radio System (STRS) Architecture Standard. Release 1.02.1
<u>NPR-2210.1</u>	Release of NASA Software
<u>STRS Website</u>	Space Telecommunications Radio Systems (STRS) Website (Password restricted but available soon.)

NASA-STD-4009

National Institute of Standards and Technology

Document Number	Document Title
FIPS PUB 140-2	Security Requirements for Cryptographic Modules

Non-Government Documents

Institute of Electrical and Electronics Engineers (IEEE)

Document Number	Document Title
IEEE 1003.1 TM -1990	IEEE Standard for Information Technology—Portable Operating System Interface (POSIX®)
IEEE Std 1003.1b TM -1993	IEEE Standard for Information Technology - Portable Operating System Interfaces (POSIX®)- Part 1: System Application Program Interface (API) - Amendment 1: Realtime Extension (C Language)
IEEE Std 1003.1c TM -1995	IEEE Standard for Information Technology--Portable Operating System Interface (POSIX®) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)
IEEE Std 1003.1d TM -1999	IEEE Standard for Information Technology - Portable Operating System Interfaces (POSIX®)- Part 1: System Application Program Interface (API) - Amendment 4: Additional Realtime Extensions (C Language)
IEEE Std 1003.1j TM -2000	IEEE Standard for Information Technology - Portable Operating System Interfaces (POSIX®)- Part 1: System Application Program Interface (API) - Amendment 5: Advanced Realtime Extensions (C Language)
IEEE Std 1003.1q TM -2000	IEEE Standard for Information Technology - Portable Operating System Interfaces (POSIX®)- Part 1: System Application Program Interface (API) - Amendment 7: Tracing (C Language)

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

NASA-STD-4009

Object Management Group (OMG)

Document Number	Document Title
ORMSC/01-07-01	Model-Driven Architecture (MDA), Object Management Group (OMG) Architecture Board ORMSC1

American National Standards Institute (ANSI), International Standards Organization (ISO), and International Electrotechnical Commission (IEC) Joint Technical Committee (JTC) 1 Working Group

Document Number	Document Title
C99RationaleV5.10	Rationale for International Standard—Programming Languages—C
ISO/IEC 9899 (In USA this is: INCITS/ISO/IEC 9899:year)	Information technology—Programming languages—C
ISO/IEC 9945-1:2003 (IEEE Std 1003.1)	Information technology—Portable Operating System Interface (POSIX®)
ISO/IEC 14882 (In USA this is: INCITS/ISO/IEC 14882:year)	Information technology—Programming languages—C++

APPROVED FOR PUBLIC RELEASE—DISTRIBUTION IS UNLIMITED

APPENDIX D

ACKNOWLEDGEMENTS

Principal Authors

Richard C. Reinhart, Thomas J. Kacpura,
Louis M. Handler, Sandra K. Johnson, Janette
C. Briones, Jennifer M. Nappier, and Joseph
A. Downey

Glenn Research Center, Cleveland, OH

Dale J. Mortensen
ASRC Aerospace Corporation, Cleveland, OH

C. Steve Hall
Analex Corporation, Cleveland, OH

James P. Lux
Jet Propulsion Laboratory, Pasadena, CA

Key Industry Participants

Carl Smith, John Liebetreu
General Dynamics Corporation, C4-I

Mark Scoville
L-3 Communications
Salt Lake City, UT

Vince Kovarik
Harris Corporation, Melbourne, FL

Jerry Bickle
Prism Tech
Woburn, MA

Key Reviewers and Contributors

David J. Israel
Goddard Space Flight Center,
Greenbelt, MD

Andrew L. Benjamin
Johnson Space Center, Houston, TX

Allen Farrington, Yong Chong, Kenneth J. Peters
Jet Propulsion Laboratory, Pasadena, CA

Eric A. Eberly, Terry M. Luttrell
Marshall Space Flight Center, Huntsville, AL

SDR Forum Contributing Member Companies

General Dynamics
Prism Tech
Boeing Corporation
Cincinnati Electronics

Harris Corporation
L-3 Communications
Lockheed Martin